MongoDB BASICS

The Building Blocks of MongoDB

MongoDB Logo

The leaf logo represents growth, scalability and flexibility – core values of MongoDB.

NoSQL Database

MongoDB is a NoSQL database, meaning it does not use traditional tables like SQL databases.

Document Database

MongoDB stores data in documents (JSON-like format) instead of rows and columns.

Collections

Collections are groups of MongoDB documents. Similar to tables in relational databases.

Documents

Documents are the basic unit of data in MongoDB. Stored in BSON format (similar to JSON).

BSON

Binary JSON (BSON) is the binary-encoded serialization format used to store documents in MongoDB.

JSON

MongoDB documents are JSON-like, making them easy to read, write and understand.

Why MongoDB?

- High Performance
- Scalable & Flexible
- Highly Available
- Cost Effective
- Developer Friendly

Flexible Schema

MongoDB has a dynamic schema. Documents in the same collection can have different fields.

MongoDB INSTALLATION & SETUP

Install • Configure • Connect • Start Building

1. MongoDB Server

- Core of MongoDB.
- Stores all your data.
- Runs as a background service.
- Handles requests from applications.

```
# Start MongoDB Server (Windows)
net start MongoDB
```

2. MongoDB Compass

- Official GUI for MongoDB.
- Visualize your data.
- Manage databases, collections and users.
- Run queries easily.

```
Download from:
https://www.mongodb.com/try/download/compass
```

3. MongoDB Shell (mongosh)

- Command line interface.
- Interact with MongoDB.
- Run commands & scripts.
- Powerful for developers.

```
# Start mongosh
mongosh
```

4. Connection Details

Host	:	localhost
Port	:	27017
Protocol	:	mongodb://
Default Port		27017

Default connection runs on **localhost** at port **27017**.

How Connection Works

Default Connection URI

```
mongodb://localhost:27017
```

Protocol | Host | Port

You can change host or port as per your setup.

Quick Installation Steps

1. **Download MongoDB Server**
   From `mongodb.com/try/download/community`
2. **Install MongoDB**
   Follow the installation wizard.
3. **Start the MongoDB Server**
   Make sure the service is running.
4. **Install MongoDB Compass (Optional)**
   Install Compass for easy GUI management.
5. **Open mongosh (MongoDB Shell)**
   Run `mongosh` in terminal or command prompt.
6. **Connect to MongoDB**
   Use connection string: mongodb://localhost:27017

Verify Installation

```
> mongosh
> show dbs
admin       40.00 KiB
config      12.00 KiB
local       40.00 KiB
>
```

MongoDB is ready to use!

Key Points

- MongoDB Server manages your data.
- MongoDB Compass provides visual interface.
- mongosh is the powerful command line shell.
- Default connection is `localhost:27017`.
- Data is stored in Databases → Collections → Documents.

MongoDB ARCHITECTURE

From Client Request to Data Storage

1. CLIENT

Applications, tools or users send requests to MongoDB.

2. MONGODB DRIVER

Driver provides an interface between the application and MongoDB server.

3. MONGODB SRIVER

MongoDB server processes requests, manages data and returns results.

4. DATABASE

myDatabase

A container server processes. Each database is isolated from others.

5. COLLECTION

users

A group of MongoDB documents. Similar to a table in RDBMS.

6. DOCUMENT

```
{
  "_id": ObjectId("..."),
  "name": "John Doe",
  "age": 25,
  "email": "john@example.com",
  "skills": ["MongoDB", "Node.js"],
  "active": true
}
```

The basic unit of data in MongoDB. Stored in BSON format.

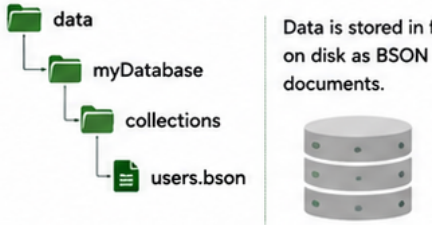
7. BSON

Binary JSON (BSON) is the format used to store documents internally.

8. STORAGE

- Data is persisted on disk in the storage engine.
- WiredTiger is the default storage engine.
- Provides high performance, concurrency and durability.

On Disk



Data is stored in files on disk as BSON documents.

Disk / SSD

FLOW SUMMARY

1. Client sends a request
2. Driver sends the request to MongoDB Server
3. Server processes the request
4. Data is accessed from the Database
5. From the appropriate Collections
6. Documents are read/written in BSON format
7. BSON data is handled by the server
8. Data is stored persistently in Storage

KEY COMPONENTS

- **Client** — Any application or tool.
- **Driver** — Language specific MongoDB driver.
- **Server** — Core of MongoDB.
- **Database** — Logical container for collections.
- **Collection** — Group of documents.
- **Document** — JSON-like data structure.
- **BSON** — Binary JSON format.
- **Storage** — Data persisted on disk.

Why This Architecture?

- Flexible Schema-less Data Model
- High Performance & Scalability
- Easy Horizontal Scaling
- High Availability & Fault Tolerance
- Built for Modern Applications & Big Data

MongoDB DATABASE & COLLECTIONS

Organize Your Data Efficiently

COMMANDS CHEAT SHEET

show dbs	Show all databases in the server.
use database	Switch to a database. Create if not exists.
show collections	Show all collections in the current database.
db.createCollection()	Create a new collection in the current database.
db.collection	Reference to a collection. Perform operations.
drop()	Drop (delete) a collection from the database.

DATABASE, COLLECTION & DOCUMENTS

DATABASE

A database is a container for collections.

Example: mydb

COLLECTION

A collection is a group of MongoDB documents.

Example: users

DOCUMENTS

```
{
  _id: ObjectId("665f..."),
  name: "John Doe",
  age: 25,
  email: "john@example.com",
  city: "New York",
  active: true
}
```

Documents are stored in BSON (Binary JSON) format.

Multiple Documents

KEY POINTS

- A database contains one or more collections.
- A collection contains many documents.
- Documents are schema-less (JSON/BSON format).
- Collections are created automatically when you insert data or explicitly using commands.
- You can drop collections when you no longer need them.

EXAMPLES (MONGOSH)

1. show dbs

```
> show dbs
admin      40.00 KiB
config     60.00 KiB
local      72.00 KiB
```

Shows all databases.

2. use database

```
> use mydb
switched to db mydb
```

Switch to (or create) database.

3. show collections

```
> show collections
users
products
orders
```

Shows collections in current database.

4. db.createCollection()

```
> db.createCollection("users")
{
   "ok" : 1
}
```

Create a new collection.

5. db.collection

```
> db.users
users
```

Reference to the collection "users".

6. drop()

```
> db.users.drop()
true
```

Drop (delete) the collection.

SUMMARY FLOW

Create or Switch Database (use database)

Create or View Collections (show/createCollection)

Insert & Manage Documents (CRUD Operations)

Drop Collection When Not Needed (drop())

MongoDB is flexible, powerful and designed for modern applications.

MongoDB DOCUMENTS

mongoDB.

The Basic Unit of Data in MongoDB

WHAT IS A DOCUMENT?

A Document is a record in MongoDB. It is a set of key-value pairs stored in **BSON** format.

Documents are stored inside **Collections**, which are stored inside **Databases**.

KEY TERMS

- **_id** — Primary key of the document. Must be unique.
- **Field** — The key (name) in a document. Example: `name`, `age`
- **Value** — The value associated with a field. Can be any data type.
- **ObjectId** — Default data type for _id. Example: `64f8a1b9e5f7c2d4a8b12345`
- **Nested Object** — An object inside another object. Also called embedded document.
- **Array** — An ordered list of values. Can store multiple items.
- **Binary JSON (BSON)** — The format used to store documents in MongoDB.

EXAMPLE DOCUMENT

Document
_id (ObjectId)
Field
Value (String)
Value (Number)
Nested Object
Array
Value (Boolean)

```
{
    _id: ObjectId("665f1a2b3c4d5e6f7a8b9c0d"),
    name: "John Doe",
    age: 25,
    email: "john@example.com",
    address: {
        street: "123 Main St",
        city: "New York",
        zip: "10001"
    },
    skills: ["MongoDB", "JavaScript", "Node.js"],
    active: true,
    createdAt: ISODate("2024-05-20T10:15:30Z")
}
```

DOCUMENT STRUCTURE OVERVIEW

Database	Collection	Document	Field (Key)	Value
Contains Collections	Contains Documents	Key-Value Pairs	Property / Key	Data / Value
			name	"John Doe"

BSON FORMAT

MongoDB stores documents in BSON (Binary JSON) format internally.

JSON → BSON { }

Efficient, compact & fast

COMMON DATA TYPES IN DOCUMENTS

Tt	String	"Hello World"
123	Number	25, 3.14
✓	Boolean	true, false
	Date	ISODate("2024-05-20")
{ }	Object (Embedded Document)	{ key: "value" }
[ ]	Array	["A", "B", "C"]
null	Null	null

IMPORTANT NOTES

- Each document must have a unique _id. If not provided, MongoDB generates one automatically.
- Documents in the same collection can have different fields. (Schema-less / Flexible)
- Field order does not matter. Fields can be added or removed anytime.
- Ideal for hierarchical data, real-time apps, and rapid development.

Database → Collection → Document

Data Organized the MongoDB Way!

MongoDB INSERT DOCUMENTS

Add Data to Your Collections

KEY POINTS

- Insert documents into a collection using insertOne() or insertMany().
- If _id is not provided, MongoDB generates it automatically.
- Documents are stored in BSON format.
- insertOne() inserts a single document.
- insertMany() inserts multiple documents.

COLLECTION

A collection is a group of MongoDB documents. Documents in a collection can have different fields.

1. insertOne() – Insert Single Document

Inserts a single document into a collection.

```
> db.users.insertOne({
    name: "John Doe",
    age: 25,
    email: "john@example.com",
    city:  "New York"
});
```

Result:

```
{ acknowledged: true, insertedId: ObjectId("665f1a2b3c4d5e6f7a8b9c0d") }
```

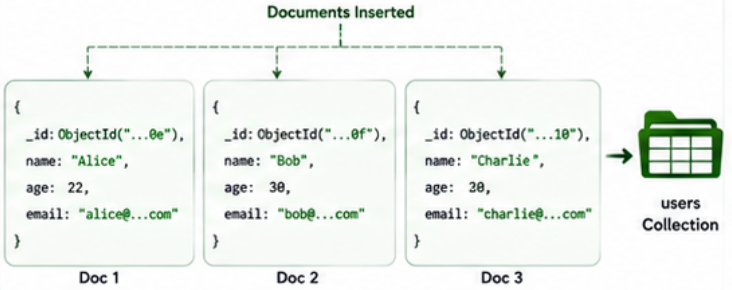
2. insertMany() – Insert Multiple Documents

Inserts multiple documents into a collection.

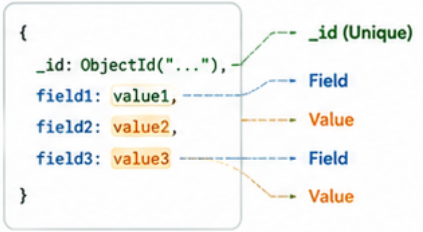
```
> db.users.insertMany([
   { name: "Alice", age: 22, email: "alice@example.com" },
   { name: "Bob", age: 30, email: "bob@example.com" },
   { name: "Charlie", age: 28, email: "charlie@example.com" }
]);
```

Result:

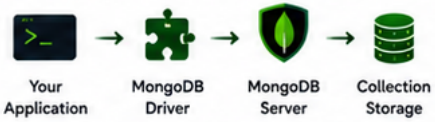
```
{
  acknowledged: true,
  insertedIds: {
    "0": ObjectId("665f1a2b3c4d5e6f7a8b9c0e"),
    "1": ObjectId("665f1a2b3c4d5e6f7a8b9c0f"),
    "2": ObjectId("665f1a2b3c4d5e6f7a8b9c10")
  }
}
```



DOCUMENT STRUCTURE



INSERT FLOW



COMMON COMMANDS

```
db.collection.insertOne(document)
// Insert single document

db.collection.insertMany([doc1, doc2])
// Insert multiple documents

db.collection.find()
// View documents

db.collection.countDocuments()
// Count documents

db.collection.drop()
// Drop collection
```

IMPORTANT NOTES

- _id must be unique in a collection.
- Fields in a document are flexible.
- Data is stored in BSON (Binary JSON).

SAMPLE COLLECTION AFTER INSERT OPERATIONS

_id	name	age	email	city
ObjectId("665f1...0d")	John Doe	25	john@example.com	New York
ObjectId("665f1...0e")	Alice	22	alice@example.com	null
ObjectId("665f1...0f")	Bob	30	bob@example.com	null
ObjectId("665f1...10")	Charlie	28	charlie@example.com	null

QUICK COMPARISON

insertOne()	insertMany()
Insert single document	Insert multiple documents
Returns insertedId	Returns insertedIds
Use for one record	Use for bulk insert

BEST PRACTICES

- Always validate data before inserting.
- Use meaningful field names.
- Keep documents small and focused.
- Use insertMany() for bulk operations for better performance.

MongoDB FIND DOCUMENTS

Search and Retrieve Data Easily

KEY CONCEPTS

find()
Returns all documents that match the query.

findOne()
Returns the first document that matches the query.

Query / Filter
Defines the conditions to match documents.

Projection
Specifies which fields to include or exclude.

Results
The documents returned by the query.

Cursor
find() returns a cursor that you can iterate.

SYNTAX

find()

```
db.collection.find(<query>, <projection>)
```

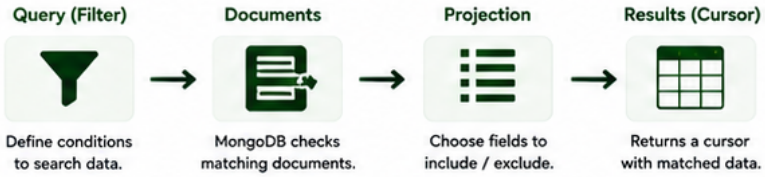
findOne()

```
db.collection.findOne(<query>, <projection>)
```

QUERY OPERATORS (EXAMPLES)

Operator	Meaning	Example
$eq	Equal to	{ age: { $oq: 25 } }
$ne	Not equal to	{ age: { $ne: 25 } }
$gt	Greater than	{ age: { $gt: 25 } }
$lt	Less than	{ age: { $lt: 25 } }
$in	In array	{ status: { $in: ["A","B"] } }
$and	AND condition	{ $and: [{ age: { $gt: 20 } }, { city: "NY" }] }
$or	OR condition	{ $or: [{ city: "NY" }, { city: "LA" }] }

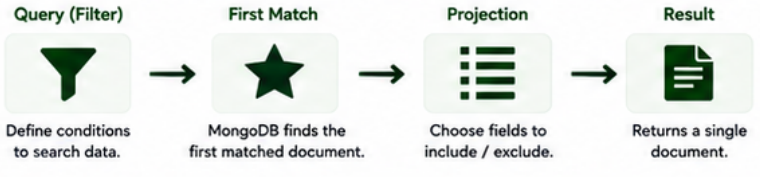
1. find() – RETURN MULTIPLE DOCUMENTS




```
// Example: All users
db.users.find();

// Example: Filter + Projection
db.users.find(
  { age: { $gt: 18 } },
  { name: 1, email: 1, _id: 0 }
);
```

2. findOne() – RETURN FIRST MATCHED DOCUMENT




```
// Example: First user with age > 25
db.users.findOne(
  { age: { $gt: 25 } },
  { name: 1, age: 1, _id: 0 }
);
```

3. HOW find() WORKS (CURSOR)

SAMPLE COLLECTION: users

```
[
  {
    "_id": ObjectId("665f1a2b3c4d5e6f7a8b9c01"),
    "name": "John", "age": 25,
    "email": "john@example.com", "city": "NY"
  },
  {
    "_id": ObjectId("665f1a2b3c4d5e6f7a8b9c02"),
    "name": "Alice", "age": 22,
    "email": "alice@example.com", "city": "LA"
  },
  {
    "_id": ObjectId("665f1a2b3c4d5e6f7a8b9c03"),
    "name": "Bob", "age": 30,
    "email": "bob@example.com", "city": "NY"
  }
]
```

RESULT EXAMPLE (findOne)

```
{
  "name": "Bob",
  "age": 30
}
```

PROJECTION EXAMPLES

Include Fields

```
db.users.find(
  { city: "NY" },
  { name: 1, email: 1, _id: 0 }  // include
);
```

Sample Result

```
[
  { "name": "John", "email": "john@example.com" },
  { "name": "Bob", "email": "bob@example.com" }
]
```

EXCLUDE FIELDS

Exclude Fields

```
db.users.find(
  {},
  { email: 0 }  // exclude email
);
```

Sample Result

```
[
  { "_id": ObjectId("...01"), "name": "John",
    "age": 25, "city": "NY" },
  { "_id": ObjectId("...02"), "name": "Alice",
    "age": 22, "city": "LA" }
]
```

QUICK REFERENCE

db.collection.find()	Get all documents
db.collection.find(filter)	Get documents by filter
db.collection.find(filter, projection)	Filter + select fields
db.collection.findOne()	Get first document
db.collection.find().sort()	Sort results
db.collection.find().limit(n)	Limit results
db.collection.find().skip(n)	Skip results

PRO TIPS

- Use projections to return only needed fields for better performance.
- Use indexes on fields used in queries for faster results.
- Use limit() and sort() with large datasets.
- Remember: find() returns a cursor. Iterate to get results.

Powerful queries. Flexible schema. Lightning fast!
That's the power of MongoDB.

mongoDB.

MongoDB QUERY OPERATORS

Powerful Operators for Advanced Queries

WHAT ARE QUERY OPERATORS?

Query operators are special keywords used in queries to filter documents based on conditions.

HOW TO USE

```
db.collection.find({
  field: { <operator>: value }
})
```

EXAMPLE COLLECTION: users

```
[
  { "_id": 1, "name": "John", "age": 25, "city": "New York"
  { "_id": 2, "name": "Alice", "age": 22, "city": "Los Angeles"
  },
  { "_id": 3, "name": "Bob", "age": 30, "city": "Chicago"
  { "_id": 4, "name": "Charlie", "age": 28, "city": "New York"
  },
  { "_id": 5, "name": "David", "age": 35, "city": "Boston" }
]
```

QUICK TIPS

- Operators are case-sensitive.
- Use indexes for better performance.
- Combine operators for complex queries.
- Test your queries in MongoDB Compass or mongosh.

OPERATOR	MEANING	SYNTAX	EXAMPLE QUERY	MATCHED DOCUMENTS
$eq	**Equal to** (Matches values that are equal to the given value.)	`{ field: { $eq: value } }` `// or { field: value }`	`db.users.find({ age: { $eq: 25} })` `// or db.users.find({ age: 25 })`	`{ "_id": 1, "name": "John", "age": 25, "city": "New York" }`

$ne	**Not equal to** (Matches values that are not equal to the given value.)	`{ field: { $ne: value } }`	`db.users.find({ age: { $ne: 25} })`	`{ "_id": 2, "name": "Alice", "age": 22, "city": "Los Angeles" }`
				`{ "_id": 3, "name": "Bob", "age": 30, "city": "Chicago" }`
				`{ "_id": 4, "name": "Charlie", "age": 28, "city": "New York" }`
				`{ "_id": 5, "name": "David", "age": 35, "city": "Boston" }`

$gt	**Greater than** (Matches values greater than the given value.)	`{ field: { $gt: value } }`	`db.users.find({ age: { $gt: 25} })`	`{ "_id": 3, "name": "Bob", "age": 30, "city": "Chicago" }`
				`{ "_id": 4, "name": "Charlie", "age": 28, "city": "New York" }`
				`{ "_id": 5, "name": "David", "age": 35, "city": "Boston" }`

$gte	**Greater than or equal to** (Matches values greater than or equal to the given value.)	`{ field: { $gte: value } }`	`db.users.find({ age: { $gte: 25 } })`	`{ "_id": 1, "name": "John", "age": 25, "city": "New York" }`
				`{ "_id": 3, "name": "Bob", "age": 30, "city": "Chicago" }`
				`{ "_id": 4, "name": "Charlie", "age": 28, "city": "New York" }`
				`{ "_id": 5, "name": "David", "age": 35, "city": "Boston" }`

$lt	**Less than** (Matches values less than the given value.)	`{ field: { $lt: value } }`	`db.users.find({ age: { $lt: 25} })`	`{ "_id": 2, "name": "Alice", "age": 22, "city": "Los Angeles" }`
$lte	**Less than or equal to** (Matches values less than or equal to the given value.)	`{ field: { $lte: value } }`	`db.users.find({ age: { $lte: 25 } })`	`{ "_id": 1, "name": "John", "age": 25, "city": "New York" }` `{ "_id": 2, "name": "Alice", "age": 22, "city": "Los Angeles" }`

$in	**In** (Matches any value that exists in an array.)	`{ field: { $in: [v1, v2, ..] } }`	`db.users.find({ city:`
			`{ $in: ["New York", "Chicago"]} })`
				`{ "_id": 1, "name": "John", "age": 25, "city": "New York" }`
				`{ "_id": 3, "name": "Bob", "age": 30, "city": "Chicago" }`
				`{ "_id": 4, "name": "Charlie", "age": 28, "city": "New York" }`

$nin	**Not in** (Matches values that do not exist in the array.)	`{ field: { $nin: [v1, v2, ..] } }`	`db.users.find({ city;`
			`{ $nin: ["New York"] } })`
				`{ "_id": 2, "name": "Alice", "age": 22, "city": "Los Angeles" }`
				`{ "_id": 3, "name": "Bob", "age": 30, "city": "Chicago" }`
				`{ "_id": 5, "name": "David", "age": 35, "city": "Boston" }`

COMBINE OPERATORS

You can combine multiple operators using logical operators.

```
db.users.find({
  age: { $gte: 25, $lte: 30 },
  city: { $in: ["New York", "Chicago"] }
})
```

LOGICAL OPERATORS

- **$and** (All conditions must match)
- **$or** (At least one condition must match)
- **$nor** (None of the conditions match)

PRO TIP

Use query operators to filter data accurately and build powerful applications!

MongoDB LOGICAL OPERATORS

Combine Multiple Conditions with Logic

WHAT ARE LOGICAL OPERATORS?

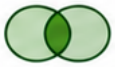
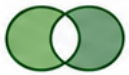
Logical operators allow you to combine multiple conditions in a query and filter documents using logic.

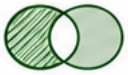
LOGICAL OPERATORS

- **$and** — Matches documents that satisfy all the conditions.
- **$or** — Matches documents that satisfy at least one condition.
- **$not** — Negates the result of a condition.
- **$nor** — Matches documents that fail all the conditions.

SYNTAX OVERVIEW

```
db.collection.find({
  <logical_operator>: [
    { condition1 },
    { condition2 }
  ]
})
```

OPERATOR	DESCRIPTION	SYNTAX	EXAMPLE QUERY	EXAMPLE RESULT
$and  All conditions must be true	Returns documents that match all the specified conditions.	`{ $and: [ { condition1 }, { condition2 }, ... ] }`	`// Find users age > 20 AND city = "New York"` `db.users.find({ $and: [ { age: { $gt: 20 } }, { city: "New York" } ] })`	_id | name | age | city 1 | John | 25 | New York 3 | Charlie | 28 | New York (All conditions matched)
$or  At least one condition must be true	Returns documents that match at least one of the specified conditions.	`{ $or: [ { condition1 }, { condition2 }, ... ] }`	`// Find users age < 20 OR city = "Los Angeles"` `db.users.find({ $or: [ { age: { $lt: 20 } }, { city: "Los Angeles" } ] })`	_id | name | age | city 2 | Alice | 18 | Chicago 4 | Bob | 30 | Los Angeles 5 | David | 22 | Los Angeles (At least one condition matched)
$not Inverts the result of a condition	Returns documents that do NOT match the specified condition.	`{ field: { $not: { <operator>: value } } }`	`// Find users where age is NOT greater than 30` `db.users.find({ age: { $not: { $gt: 30 } } })`	_id | name | age | city 1 | John | 25 | New York 2 | Alice | 18 | Chicago 5 | David | 22 | Los Angeles (age is NOT > 30)

$nor  None of the conditions must be true	Returns documents that do NOT match any of the specified conditions.	`{ $nor: [ { condition1 }, { condition2 }, ... ] }`	`// Find users who are NOT in New York`
			`// AND NOT older than 30`
			`db.users.find({ $nor: [ { city: "New York" }, { age: { $gt: 30 } } ] })`
				_id | name | age | city
				2 | Alice | 18 | Chicago
				(None of the conditions matched)

PRO TIP

- Use indexes on fields used in conditions.
- Keep queries simple and efficient.
- Test queries in MongoDB Compass or mongosh.

MORE EXAMPLES

DESCRIPTION	QUERY
Age > 20 AND city = "Chicago"	`db.users.find({ $and: [ { age: { $gt: 20 } }, { city: "Chicago" } ] })`
Age < 20 OR age > 30	`db.users.find({ $or: [ { age: { $lt: 20 } }, { age: { $gt: 30 } } ] })`
City is NOT "New York"	`db.users.find({ city: { $not: { $eq: "New York" } } })`
NOT (city = "Chicago" OR age < 18)	`db.users.find({ $nor: [ { city: "Chicago" }, { age: { $lt: 18 } } ] })`

CHOOSE THE RIGHT OPERATOR

- Use $and when all conditions must match.
- Use $or when any one condition can match.
- Use $not to negate a single condition.
- Use $nor to exclude documents that match any of the conditions.

VISUAL SUMMARY

- **$and** All conditions must be true
- **$or** At least one condition is true
- **$not** Negates a condition
- **$nor** None of the conditions are true

MongoDB UPDATE DOCUMENTS

Modify Existing Data in Your Collections

WHAT IS UPDATE?

Update operations modify existing documents in a collection based on a filter (condition).

UPDATE METHODS

updateOne()
Updates the first document that matches the filter.

updateMany()
Updates all documents that match the filter.

SYNTAX

```
db.collection.updateOne(
   <filter>,
   { $operator: { field: value } },
   { options }
)

db.collection.updateMany(
   <filter>,
   { $operator: { field: value } },
   { options }
)
```

1. UPDATE METHODS EXAMPLE

updateOne() Example

```
// Increase age of first user named "John"
db.users.updateOne(
  { name: "John" },
  { $set: { age: 26 } }
)
```

Before

```
{ _id: 1, name: "John", age: 25 }
```

↓

After

```
{ _id: 1, name: "John", age: 26 }
```

updateMany() Example

```
// Increase age of all users from "NY"
db.users.updateMany(
  { city: "NY" },
  { $inc: { age: 1 } }
)
```

Before

```
{ _id: 1, name: "John",  age: 25, city: "NY" }
{ _id: 2, name: "Alice", age: 22, city: "NY" }
```

↓

After

```
{ _id: 1, name: "John",  age: 26, city: "NY" }
{ _id: 2, name: "Alice", age: 23, city: "NY" }
```

3. COMBINING OPERATORS

```
// Update multiple fields at once
db.users.updateOne(
  { name: "John" },
  {
    $set: { city: "Los Angeles" },
    $inc: { age: 1 },
    $unset: { email: "" }
  }
)
```

Before

```
{ _id: 1, name: "John", age: 25,
 city: "NY", email: "john@example.com" }
```

↓

After

```
{ _id: 1, name: "John", age: 26,
 city: "Los Angeles" }
```

2. UPDATE OPERATORS

Operator	Description	Syntax	Example	Result
$set	Sets the value of a field.	`{ $set: { field: value } }`	`db.users.updateOne( { name: "John" }, { $set: { age: 26 } } )`	Sets age to 26
$unset	Removes the specified field.	`{ $unset: { field: "" } }`	`db.users.updateOne( { name: "John" }, { $unset: { email: "" } } )`	Removes email field
$rename	Renames a field.	`{ $rename: { oldName: newName} }`	`db.users.updateOne( { name: "John" }, { $rename: { fullname: "name" } } )`	Renames fullname to name
$inc	Increments the value of a field.	`{ $inc: { field: number } }`	`db.users.updateOne( { name: "John" }, { $inc: { age: 1 } } )`	Increases age by 1
$mul	Multiplies the value of a field.	`{ $mul: { field: number } }`	`db.products.updateOne( { name: "Laptop" }, { $mul: { price: 1.10 } } )`	Multiplies price by 1.10 (10%)
$min	Updates the field only if the value is less.	`{ $min: { field: value } }`	`db.users.updateOne( { name: "John" }, { $min: { age: 20 } } )`	Sets age to 20 only if < 20
$max	Updates the field only if the value is greater.	`{ $max: { field: value } }`	`db.users.updateOne( { name: "John" }, { $max: { age: 30 } } )`	Sets age to 30 only if > 30

OPTIONS (3rd PARAMETER)

Option	Type	Description	Example
upsert	Boolean	Insert document if no match found	`{ upsert: true }`
multi	Boolean	Update multiple documents (default for updateMany)	`{ multi: true }`
arrayFilters	Array	Filter elements in an array (advanced)	`{ arrayFilters: [{ "elem.x": 5 }] }`

PRACTICAL EXAMPLES

```
db.users.updateOne({ name: "John" }, { $set: { age: 26 } })          // Set age to 26
db.users.updateMany({ city: "NY" }, { $inc: { loginCount: 1 } })     // Increment loginCount
db.users.updateOne({ name: "John" }, { $unset: { email: "" } })      // Remove email field
db.users.updateOne({ name: "John" }, { $rename: { fullname: "name" } }) // Rename field
db.products.updateOne({ name: "Laptop" }, { $mul: { price: 1.10 } }) // Increase price by 10%
db.users.updateOne({ name: "John" }, { $min: { age: 20 } })          // Set age to min 20
db.users.updateOne({ name: "John" }, { $max: { age: 30 } })          // Set age to max 30
```

BEST PRACTICES

- Always use specific filters to avoid unintended updates.
- Use updateOne() when only one document needs to be updated.
- Use updateMany() carefully on large collections.
- Test your updates in MongoDB Compass before running in production.

PRO TIP

Always backup your data before performing bulk updates!

MongoDB ARRAY OPERATORS

Array Update Operators for Efficient Array Manipulation

WHAT ARE ARRAY OPERATORS?

Array operators help you update, modify and manipulate array fields inside documents easily.

COMMON ARRAY UPDATE METHODS

updateOne()
Updates the first document that matches the filter.

updateMany()
Updates all documents that match the filter.

Find + Update
Find documents and then apply array updates.

SYNTAX OVERVIEW

```
db.collection.updateOne(
  { <filter> },
  { <update: { <operator>: { <field>: value } } },
  { <options> }
)

# For updateMany() use same syntax
# Works with all array operators
```

ARRAY OPERATORS

OPERATOR	DESCRIPTION	SYNTAX	EXAMPLE QUERY	EXAMPLE RESULT
$push	Adds an element to the end of an array.	`{ $push: { field: value } }` `{ $push: { field: { $each: [] } } }`	`db.users.updateOne( { _id: 1 }, { $push: { skills: "MongoDB" } } )`	Before: `skills: ["HTML", "CSS"]` → After: `skills: ["HTML", "CSS", "MongoDB"]`
$pop	Removes the first (-1) or last (1) element from an array.	`{ $pop: { field: 1 } } // last` `{ $pop: { field: -1 } } // first`	`db.users.updateOne( { _id: 1 }, { $pop: { skills: -1 } } )`	Before: `skills: ["HTML", "CSS", "MongoDB"]` → After: `skills: ["CSS", "MongoDB"]`
$pull	Removes all array elements that match the given value(s).	`{ $pull: { field: value } }` `{ $pull: { field: { $in: [] } } }`	`db.users.updateOne( { _id: 1 }, { $pull: { skills: "CSS" } } )`	Before: `skills: ["HTML", "CSS", "MongoDB"]` → After: `skills: ["HTML", "MongoDB"]`
$addToSet	Adds a value to an array only if the value does not already exist.	`{ $addToSet: { field: value } }` `{ $addToSet: { field: { $each: [] } } }`	`db.users.updateOne( { _id: 1 }, { $addToSet: { skills: "MongoDB" } } )`	Before: `skills: ["HTML", "CSS"]` → After: `skills: ["HTML", "CSS", "MongoDB"]` (No duplicate if "MongoDB" already exists)
$each	Used with $push or $addToSet to add multiple elements.	`{ $push: { field: { $each: [v1, v2, ...] } } }` `{ $addToSet: { field: { $each: [v1, v2, ...] } } }`	`db.users.updateOne( { _id: 1 }, { $push: { skills: { $each: ["Python", "Node.js"] } } } )`	Before: `skills: ["HTML"]` → After: `skills: ["HTML", "Python", "Node.js"]`
$slice	Used with $push to limit the size of an array.	`{ $push: { field: { $each: [], $slice: n } } }` `// keeps only last n elements`	`db.users.updateOne( { _id: 1 }, { $push: { activities: { $each: ["A"], $slice: -3 } } } )`	Before: `activities: ["A","B","C","D","E"]` → After: `activities: ["C","D","E"]` (keeps last 3)
$position	Used with $push to insert an element at a specific position.	`{ $push: { field: { $each: [value], $position: n } } }` `// n = index position`	`db.users.updateOne( { _id: 1 }, { $push: { skills: { $each: ["MongoDB"], $position: 1 } } } )`	Before: `skills: ["HTML", "CSS", "JS"]` → After: `skills: ["HTML", "MongoDB", "CSS", "JS"]` (inserted at index 1)

ARRAY UPDATE EXAMPLES

1. $push with $each

```
db.users.updateOne(
  { _id: 1 },
  { $push: { skills: { $each: ["Python", "JS"] } } }
)
```

2. $pull with $in

```
db.users.updateMany(
  {},
  { $pull: { skills: { $in: ["OldSkill", "CSS"] } } }
)
```

3. $addToSet with $each

```
db.users.updateOne(
  { _id: 1 },
  { $addToSet: { skills: { $each: ["Go", "Rust"] } } }
)
```

4. $push with $slice

```
db.logs.updateOne(
  { _id: 1 },
  { $push: { logs: { $each: ["L6"], $slice: -5 } } }
)
```

5. $push with $position

```
db.users.updateOne(
  { _id: 1 },
  { $push: { skills: { $each: ["MongoDB"], $position: 0 } } }
)
```

QUICK REFERENCE

Operator	Use For	Adds	Removes	Duplicates Allowed?
$push	Add to end	Yes	No	Yes
$pop	Remove first/last	No	Yes	—
$pull	Remove by value	No	Yes	—
$addToSet	Add if not exists	Yes	No	No
$each	Add multiple	Yes	No	Depends
$slice	Limit array size	Yes	Yes	—
$position	Insert at index	Yes	No	Yes

PRACTICAL EXAMPLE (COMBINATION)

```
db.users.updateOne(
  { _id: 1 },
  {
    $push: {
      skills: {
        $each: ["Go", "Rust"],
        $position: 1,
        $slice: 5
      }
    }
  }
)
```

Before: `skills: ["HTML", "CSS", "JS", "PHP"]`

Insert ["Go","Rust"] at position 1 Keep last 5 →

After: `skills: ["HTML", "Go", "Rust", "CSS", "JS"]`

BEST PRACTICES

- Use $addToSet to avoid duplicates.
- Use $push with $each for adding multiple values.
- Use $slice to keep array size under control.
- Validate array size if it grows large.
- Test array updates in MongoDB Compass first.

PRO TIP

Combine operators for powerful updates!
Example: $push + $each + $slice + $position for advanced array control.

MongoDB DELETE DOCUMENTS

Remove Data from Your Collection Safely

WHAT IS DELETE?

Delete operations remove documents that match a specified filter (query) from a collection.

DELETE METHODS

deleteOne()
Deletes the first document that matches the filter.

deleteMany()
Deletes all documents that match the filter.

SYNTAX OVERVIEW

```
db.collection.deleteOne(
   <filter>
)
 // Deletes the first matching document
 db.collection.deleteMany(
   <filter>
)
 // Deletes all matching documents
```

1. DELETE METHODS EXAMPLE

deleteOne() Example

```
// Delete the first user with name "Alice"
db.users.deleteOne(
  { name: "Alice" }
)
```

Before (Collection: users)

_id	name	age	city
1	John	25	New York
2	Alice	22	Chicago
3	Bob	30	Los Angeles
4	Alice	28	Boston

→

After (deleteOne)

_id	name	age	city
1	John	25	New York
3	Bob	30	Los Angeles
4	Alice	28	Boston

deleteMany() Example

```
// Delete all users from "Chicago"
db.users.deleteMany(
  { city: "Chicago" }
)
```

Before (Collection: users)

_id	name	age	city
1	John	25	New York
2	Alice	22	Chicago
3	Bob	30	Chicago
4	David	26	Boston
5	Emma	24	Chicago

→

After (deleteMany)

_id	name	age	city
1	John	25	New York
4	David	26	Boston

2. COMMON FILTER (QUERY) EXAMPLES

Filter (Query)	Matches
{}	All documents
{ name: "John" }	Documents where name is John
{ age: { $gt: 25 } }	Documents where age > 25
{ city: "Chicago" }	Documents where city is Chicago
{ age: { $lte: 25 }, city: "NY" }	Age ≤ 25 AND city is NY
{ status: "inactive" }	Documents where status is inactive

3. DELETE WITH FILTER EXAMPLES

Operation	Query (Filter)	Example Code	Result
Delete first user older than 30	{ age: { $gt: 30 } }	db.users.deleteOne({ age: { $gt: 30 } })	Deletes only one matching document
Delete all inactive users	{ status: "inactive" }	db.users.deleteMany({ status: "inactive" })	Deletes all matching documents
Delete users from Boston with age < 25	{ city: "Boston", age: { $lt: 25 } }	db.users.deleteMany({ city: "Boston", age: { $lt: 25 } })	Deletes all matching documents

4. DELETE ALL DOCUMENTS (USE WITH CAUTION)

If you use an empty filter {} with deleteMany(), it will delete ALL documents in the collection.

```
db.users.deleteMany({})
```

→ All documents will be deleted!

SAFETY TIPS

- Always use a filter to avoid deleting all data.
- Double-check your query before running.
- Use deleteOne() when only one match is expected.
- Use deleteMany() when multiple matches are intended.
- Backup important data before bulk delete.

5. OPERATION FLOW

Collection (users) → Query / Filter (Find what to delete) → Match Documents (Apply filter) → Remove Data (Delete) → Result (Modified Count)

6. RESULT & OUTPUT

```
> db.users.deleteMany({ city: "Chicago" })
{
  acknowledged: true,
  deletedCount: 3
}
```

acknowledged : Operation status
`true` = success

deletedCount : Number of documents deleted

QUICK REFERENCE

deleteOne(filter)	Delete first matching document
deleteMany(filter)	Delete all matching documents
{}	Matches all documents (use with caution)
Result.deletedCount	Shows how many documents were deleted

PRO TIP Test your query using find() before delete.
Example: `db.users.find({ age: { $gt: 30 } })`

CAUTION Always verify your filter. Using {} with deleteMany() will remove all documents from the collection.

BEST PRACTICE
Use specific filters, test queries, and backup data before delete operations.

MongoDB Compass
Use MongoDB Compass to visually build and test delete queries.

MongoDB PROJECTION

Return Only the Fields You Need

WHAT IS PROJECTION?

Projection controls which fields are returned in the result set of a query.
It helps to reduce network traffic and improve performance.

PROJECTION BASICS

1 Include Fields
Return only the fields specified (1).

0 Exclude Fields
Exclude the fields specified (0).

SYNTAX

```
db.collection.find(
  <filter>,
  { <field1>: <1 | 0>, <field2>: <1 | 0>, ... }
)
1 = Include the field
0 = Exclude the field
```

Note: You cannot mix `1` and `0` in the same projection (except for _id).

QUICK TIPS

- By default, all fields are returned (no projection).
- Include only the fields you need.
- Exclude large or unnecessary fields.
- _id is included by default. Exclude it if not needed.

SAMPLE COLLECTION: users

```
db.users.find()  // Default: returns all fields
```

_id	name	age	email	city	status	skills
1	John	25	john@example.com	New York	active	["JavaScript","Node.js"]
2	Alice	22	alice@example.com	Chicago	inactive	["Python","Django"]
3	Bob	30	bob@example.com	Los Angeles	active	["MongoDB","Express"]

1. INCLUDE FIELDS (Use 1)

Return only the fields you specify.

```
db.users.find({}, { name: 1, age: 1, city: 1, _id: 0 })
```

Result (Selected Data)

name	age	city
John	25	New York
Alice	22	Chicago
Bob	30	Los Angeles

Key Points

- 1 includes only the specified fields.
- _id is excluded here (_id: 0).
- Only selected data is returned.

2. EXCLUDE FIELDS (Use 0)

Return all fields except the ones you exclude.

```
db.users.find({}, { email: 0, skills: 0 })
```

Result (Selected Data)

_id	name	age	city	status
1	John	25	New York	active
2	Alice	22	Chicago	inactive
3	Bob	30	Los Angeles	active

Key Points

- 0 excludes the specified fields.
- All other fields are returned.
- _id is included by default.

3. ONLY _id FIELD

Return only the _id field.

```
db.users.find({}, { _id: 1 })
```

Result (Selected Data)

_id
1
2
3

Key Points

- Only _id field is returned.
- Useful for fetching just document IDs.

4. IMPORTANT RULES

⚠ You cannot mix 1 and 0 in the same projection.

```
{ name: 1, age: 0 } ✗
```

✓ You can exclude _id while including other fields.

```
{ name: 1, age: 1, _id: 0 } ✓
```

✓ If no projection is specified, all fields are returned.

```
db.users.find({}) ✓
```

5. PROJECTION QUICK REFERENCE

Use Case	Projection	Description	Returns
All fields (default)	`db.users.find()`	No projection specified	All fields
Include specific fields	`{ name: 1, age: 1 }`	Include name and age	Only name, age (+ _id)
Include & exclude _id	`{ name: 1, _id: 0 }`	Include name, exclude _id	Name only
Exclude specific fields	`{ email: 0, skills: 0 }`	Exclude email and skills	All others
Only _id field	`{ _id: 1 }`	Return only _id	_id only

6. PRACTICAL EXAMPLES

```
db.users.find()                                  // All fields
db.users.find({}, { name: 1, age: 1 })           // Include name and age
db.users.find({}, { name: 1, age: 1, _id: 0 })   // Include name, age (exclude _id)
db.users.find({}, { email: 0, skills: 0 })       // Exclude email and skills
db.users.find({}, { status: 0, _id: 0 })         // Exclude status and _id
db.users.find({}, { _id: 1 })                    // Only _id
```

PRO TIP

Use projection to fetch only the data you need. It improves performance, especially with large documents or slow networks.

NOTE

Projection only affects the fields returned in the result. It does not affect the actual documents in the collection.

MongoDB SORTING & LIMITING

Control the Order and Quantity of Your Results

WHAT IS SORTING & LIMITING?

Sorting arranges the documents in a specific order.
Limiting controls how many documents are returned.

QUICK OVERVIEW

sort()
Sorts documents
(1 = ASC, -1 = DESC)

limit()
Limits the number of documents returned

skip()
Skips a number of documents

SORT ORDER

1	ASCENDING (A → Z, 0 → 9)	↑
-1	DESCENDING (Z → A, 9 → 0)	↓

SYNTAX

```
db.collection.find(filter)
    .sort({ field: 1 | -1, field2: 1 | -1 })
    .skip(n)    // optional
    .limit(n)   // optional
```

QUICK TIPS

- Always use sort() before limit() for correct results.
- Use indexes on fields used in sort() for better performance.
- Use skip() carefully on large collections (can be slow).
- Use a unique field (like _id) in sort() for consistent order.

1. SORT() – ORRER YOUR RESULTS

ASCENDING (1)

```
db.users.find().sort({ age: 1 })
```

Sort by age in ascending order (smallest to largest)

_id	name	age	city
4	David	20	Boston
2	Alice	22	Chicago
1	John	25	New York
3	Bob	30	Los Angeles
5	Emma	35	Seattle

DESCENDING (-1)

```
db.users.find().sort({ age: -1 })
```

Sort by age in descending order (largest to smallest)

_id	name	age	city
5	Emma	35	Seattle
3	Bob	30	Los Angeles
1	John	25	New York
2	Alice	22	Chicago
4	David	20	Boston

MULTIPLE FIELDS

```
db.users.find().sort({ city: 1, age: -1 })
```

Sort by city ascending, then age descending

_id	name	age	city
2	Alice	35	Chicago
4	David	20	Boston
5	Emma	35	Seattle
2	John	25	New York
3	Bob	30	Los Angeles

↑ 1 = ASC (Ascending) Smallest → Largest, A → Z

↓ -1 = DESC (Descending) Largest → Smallest, Z → A

2. LIMIT() – LIMIT THE NUMBER OF RESULTS

```
db.users.find().sort({ age: 1 }).limit(3)
```

Return only first 3 documents (after sorting by age ASC)

_id	name	age	city
4	David	20	Boston
2	Alice	22	Chicago
1	John	25	New York

3. SKIP() – SKIP A NUMBER OF DOCUMENTS

```
db.users.find().sort({ age: 1 }).skip(2)
```

Skip first 2 documents (after sorting by age ASC)

_id	name	age	city
1	John	25	New York
3	Bob	30	Los Angeles
5	Emma	35	Seattle

4. SKIP() + LIMIT() – PAGING RESULTS

```
db.users.find().sort({ age: 1 }).skip(2).limit(2)
```

Skip first 2, then return next 2 documents

_id	name	age	city
1	John	25	New York
3	Bob	30	Los Angeles

5. PAGINATION EXAMPLE

```
db.users.find().sort({ _id: 1 })
    .skip((page - 1) * pageSize)
    .limit(pageSize)
```

Page	Query (pageSize = 2)
Page 1	skip(0).limit(2)
Page 2	skip(2).limit(2)
Page 3	skip(4).limit(2)

Pagination Flow

Query → Sort() → Skip() → Limit() → Results (Cursor)

6. RESULTS & CURSOR

- Find queries return a cursor.
- Sorting, skipping and limiting are applied on the cursor.
- The cursor then returns the final results.

Query → Cursor → Results

EXAMPLE COLLECTION: users

_id	name	age	city	score
1	John	25	New York	85
2	Alice	22	Chicago	90
3	Bob	30	Los Angeles	75
4	David	20	Boston	80
5	Emma	35	Seattle	95

PRACTICAL EXAMPLES

```
db.users.find().sort({ name: 1 })                    // Sort by name A → Z
db.users.find().sort({ name: -1 })                   // Sort by name Z → A
db.users.find().sort({ age: -1 }).limit(2)           // Top 2 oldest users
db.users.find().sort({ score: -1 }).limit(3)         // Top 3 highest score
db.users.find().sort({ city: 1, age: -1 })           // City ASC, Age DESC
db.users.find().sort({ _id: 1 }).skip(2).limit(2)    // Page 2 (pageSize=2)
```

BEST PRACTICES

- Use indexes on fields used in sort().
- Apply sort() before skip() and limit().
- Avoid large skip() on big collections.
- Use range queries + limit() for better pagination.
- Always test with explain() for performance.

PRO TIP

For large datasets, prefer range-based pagination (using last _id or date) instead of large skip().

Example: { _id: { $gt: lastId } }

Remember: 1 = ASC (Ascending) -1 = DESC (Descending) • Use sort() + skip() + limit() together for powerful pagination!

MongoDB DATA TYPES

Understand the Different Types of Data You Can Store

WHAT ARE DATA TYPES?

Data types define the kind of value a field can hold in MongoDB. MongoDB is schema-less, so different documents in the same collection can have different data types.

KEY POINTS

- MongoDB is schema-less.
- Fields in a document can have different data types.
- Choose the right data type for better performance and accuracy.
- Use correct types for indexing and querying.

SYNTAX EXAMPLE

```
db.collection.insertOne({
  name: "Alice",            // String
  age: 25,                  // Number (Int32)
  isActive: true,           // Boolean
  createdAt: new Date(),    // Date
  hobbies: ["Reading","Music"],    // Array
  address: { city: "Delhi", zip: 110001 }, // Object
  userId: ObjectId(),       // ObjectId
  middleName: null          // Null
})
```

MONGODB DATA TYPES OVERVIEW

#	DATA TYPE	DESCRIPTION	EXAMPLE VALUE	BSON TYPE	NOTES
1	String	A sequence of characters (string of text).	`"Hello MongoDB"`	String	Text data. Max size is 16 MB.
2	Number	Numeric values. Can be Int32, Int64, Double, Decimal128.	`42 , 3.14 , -100`	Int32 / Int64 / Double / Decimal128	Use appropriate type for the value.
3	Boolean	Logical values: true or false.	`true , false`	Boolean	Used for true/false conditions.
4	Date	Date and time value.	`ISODate("2024-05-25T10:30:00Z")`	Date	Stored in UTC (ISO 8601).
5	Array	An ordered list of values. Can contain different types.	`[ 25, "MongoDB", true, null ]`	Array	Zero or more elements.
6	Object	Embedded document with key-value pairs.	`{ city: "Delhi", zip: 110001 }`	Object	Also called embedded document.
7	ObjectId	Unique identifier for documents.	`ObjectId("665c0f15a3b2c1d9e8f7a123")`	ObjectId	Automatically created for _id field.
8	Null	Represents a null value or no value.	`null`	Null	Field exists but has no value.

EXAMPLE DOCUMENT (USING ALL TYPES)

```
{
  _id: ObjectId("665c0f15a3b2c1d9e8f7a123"),
  name: "Alice",
  age: 25,
  isActive: true,
  createdAt: ISODate("2024-05-25T10:30:00Z"),
  hobbies: ["Reading", "Music", "Travel"],
  address: {
    city: "Delhi",
    zip: 110001
  },
  middleName: null
}
```

NUMBER TYPES IN MONGODB

TYPE	DESCRIPTION	EXAMPLE
Int32	32-bit signed integer (-2^31 to 2^31-1)	25
Int64	64-bit signed integer (-2^63 to 2^63-1)	`Long("9000000000")`
Double	64-bit floating point number	3.14159
Decimal128	High precision decimal (34 digits)	`NumberDecimal("19.99")`

TYPE COMPARISON (EXAMPLE)

TYPE	EQUAL TO (==)	NOT EQUAL (!=)
25 (Int32)	25 (Int32)	"25" (String)
true (Boolean)	true (Boolean)	false (Boolean)
null (Null)	null (Null)	0 / "" / false

PRO TIP

Use the correct data type from the start to avoid conversion issues and improve query performance.

QUICK REFERENCE

TYPE		USE CASE	
String	Names, titles, descriptions	Array	List of items, tags, ids
Number	Age, price, quantity, count	Object	Address, profile, settings
Boolean	Status (active/inactive), flags	ObjectId	Primary key (_id field)
Date	Timestamps, events, logs	Null	Unknown or not available

CHECK DATA TYPE

```
// Check type of a field
db.collection.findOne({}, { field: { $type: true } })

// Example output
{ "field": { "$type": "string", "value": "Hello" } }
```

USEFUL $TYPE CODES

TYPE	CODE	TYPE	CODE	TYPE	CODE	TYPE	CODE
Double	1	Binary Data	5	Date	9	Int32	16
String	2	Undefined	6	Null	10	Int64	18
Object	3	ObjectId	7	Regular Expression	11	Decimal128	19
Array	4	Boolean	8	JavaScript	13		

MongoDB is schema-less, not type-less! Using the right data type makes your data consistent, queries faster and your application better.

BEST PRACTICE: Plan your schema and use appropriate data types for each field.

MongoDB EMBEDDED DOCUMENTS

Store Related Data Together in a Single Document

WHAT ARE EMBEDDED DOCUMENTS?

Embedded Documents (Nested Documents) allow you to store related data inside a single document. They help represent relationships in a natural, hierarchical way.

KEY CONCEPTS

Parent
The main document that contains the embedded document.

Child
The embedded (nested) document inside the parent document.

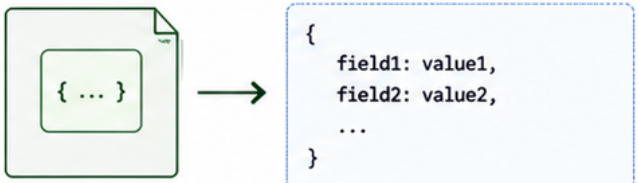
Field
A key in the document that holds a value (or an embedded object).

Object
A group of key-value pairs treated as a single unit.

PRO TIP

Use Embedded Documents for One-to-One relationships and when the embedded data is always accessed with the parent.

DOCUMENT STRUCTURE OVERVIEW



- **Parent** — Contains the embedded object
- **Child** — The embedded (nested) object
- **Field** — Key inside the object
- **Object** — Collection of key-value pairs

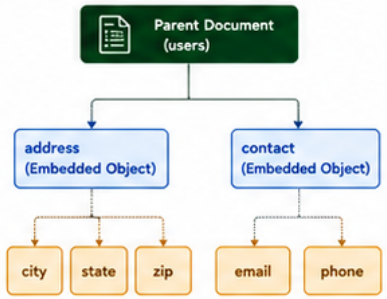
Relationship Type: One-to-One | Embedded data belongs to only one parent document.

EXAMPLE

1. INSERT DOCUMENT (WITH EMBEDDED OBJECT)

```
db.users.insertOne({
  _id: ObjectId("665c0f15a3b2c1d9e8f7a123"),
  name: "Alice",
  age: 25,
  address: {
    city: "Delhi",
    state: "Delhi",
    zip: 110001
  },
  contact: {
    email: "alice@example.com",
    phone: "9876543210"
  }
});
```

2. DOCUMENT STRUCTURE (VISUAL)



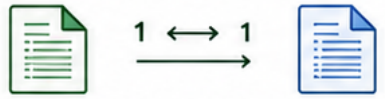
3. QUERY EXAMPLE

```
// Find user by embedded field
db.users.find({ "address.city": "Delhi" })

// Projection example
db.users.find(
  {},
  { name: 1, "address.city": 1, _id: 0 }
)

// Update embedded field
db.users.updateOne(
  { _id: ObjectId("665c0f15a3b2c1d9e8f7a123") },
  { $set: { "address.zip": 110005 } }
)
```

ONE-TO-ONE RELATIONSHIP



Each parent has one embedded child.
Each child belongs to one parent.

USE CASES

- User profile with address, contact info
- Product with specifications
- Order with payment details
- Blog post with author details

EMBEDDED vs REFERENCED

Embedded (Nested)	Referenced (Normalized)
Stored inside parent document	Stored in separate document (linked by _id)
Better read performance	Better for large / frequently changing data
One-to-One relationship	One-to-Many relationship
Denormalized	Normalized

KEY POINTS

- Embedded documents keep related data together.
- Great for One-to-One relationships.
- Improves read performance and reduces joins.
- Avoid embedding large or frequently changing data. Document size limit is 16 MB.

DOCUMENT STRUCTURE (JSON VIEW)

```
{
  _id: ObjectId("665c0f15a3b2c1d9e8f7a123"),
  name: "Alice",
  address: { city: "Delhi", state: "Delhi", zip: 110001 },
  contact: { email: "alice@example.com", phone: "9876543210" }
}
```

BEST PRACTICES

- Embed data that is accessed together.
- Keep embedded documents small and consistent.
- Use references when data grows independently.
- Design schema based on your query patterns.

MongoDB ARRAYS

Store Multiple Values in a Single Field

WHAT IS AN ARRAY?

An Array is a special data type that can store multiple values in a single field. The values can be of same or different types.

ARRAY ELEMENTS CAN BE

Strings
`["Apple", "Banana", "Mango"]`

Numbers
`[10, 20, 30, 40]`

Objects
`[{ name: "Alice", age: 25 }, { name: "Bob", age: 30 }]`

Nested Arrays
`[[1, 2], [3, 4], [5, 6]]`

Mixed Types
`[ "Hello", 10, true, null, { id: 1 } ]`

QUICK FACTS

- Arrays are 0-indexed.
- Duplicate values are allowed.
- Order of elements is important.
- Arrays can grow in size dynamically.

PRO TIP

Arrays are very powerful in MongoDB. Use them for lists, tags, categories, multiple addresses, skills, and more.

1. EXAMPLE DOCUMENT WITH ARRAYS

```
{
  _id: ObjectId("665c0f15a3b2c1d9e8f7a123"),
  name: "John",
  age: 25,
  skills: ["HTML", "CSS", "JavaScript", "MongoDB"],
  scores: [85, 90, 78, 92],
  addresses: [
    { city: "Delhi", type: "Home" },
    { city: "Mumbai", type: "Office" }
  ],
  matrix:  [[1, 2], [3, 4], [5, 6]]
}
```

DOCUMENT BREAKDOWN

Field	Type	Description
skills	Array (Strings)	List of skills
scores	Array (Numbers)	List of marks
addresses	Array (Objects)	List of address objects
matrix	Array (Nested Arrays)	2D array (matrix)

NOTE
Arrays can store any combination of strings, numbers, objects, arrays, etc.

2. ARRAY TYPES OVERVIEW

Strings Array	`["HTML", "CSS", "JS"]`
Numbers Array	`[10, 20, 30, 40]`
Objects Array	`[{ name: "A", age: 20 }, { name: "B", age: 25 }]`
Nested Arrays	`[[1, 2], [3, 4], [5, 6]]`
Mixed Array	`[ "Text", 10, false, null, { id: 1 } ]`

3. QUERY ARRAYS

Use Case	Query	Description
Match array contains value	`db.users.find({ skills: "MongoDB" })`	Find documents where array contains "MongoDB"
Match array contains any value (in list)	`db.users.find({ skills: { $in: ["HTML", "JS"] } })`	Find where array contains any of the given values
Match all values (in list)	`db.users.find({ skills: { $all: ["HTML", "CSS"] } })`	Find where array contains all the given values
Match array size	`db.users.find({ skills: { $size: 3 } })`	Find where array has exactly 3 elements
Match using index	`db.users.find({ "scores.0": { $gt: 80 } })`	Find where first element of scores > 80

4. UPDATE ARRAYS

Operation	Example	Description
$push	`db.users.updateOne( { name: "John" }, { $push: { skills: "React" } } )`	Add element at the end
$push + $each	`db.users.updateOne( { name: "John" }, { $push: { skills: { $each: ["Node.js", "Express"] } } )`	Add multiple elements
$addToSet	`db.users.updateOne( { name: "John" }, { $addToSet: { skills: "MongoDB" } }`	Add element if not exists
$pull	`db.users.updateOne( { name: "John" }, { $pull: { skills: "CSS" })`	Remove matching element
$pop	`db.users.updateOne( { name: "John" }, { $pop: { scores: 1 } }`	Remove last element (1) or first (-1)

5. NESTED ARRAYS EXAMPLE

```
db.users.find({  "matrix.1.0": 3 })
// Find where second row, first column = 3
```

Matrix Field:

```
"matrix": [[1, 2], [3, 4], [5, 6]]
```

Indexes:

```
 Row  Index  Column Index
  ↓    ↓      ↓    ↓
[ ✦ [0]  [0],  [0]  [1]    ] → Row 0
[ ✦ [1]  [0],  [1]  [1]    ] → Row 1
[ ✦ [2]  [0],  [2]  [1]    ] → Row 2
```

6. ARRAY OPERATORS QUICK REFERENCE

$in	Match any value in array	$elemMatch	Match conditions on array elements
$all	Match all values in array	$push	Add element to array
$nin	Match none of the values	$pull	Remove element from array
$size	Match array size	$addToSet	Add element if not already exists

7. PRACTICAL EXAMPLE

```
// Find users who know either HTML or JavaScript
db.users.find({ skills: { $in: ["HTML", "JavaScript"] } })
// Add multiple skills
db.users.updateOne(
  { name: "John" },
  { $push: { skills: { $each: ["Node.js", "Express"] }} }
)
// Remove CSS from skills
db.users.updateOne({ name: "John" },  { $pull: { skills: "CSS" } })
```

8. BEST PRACTICES

- Use arrays for related items (tags, skills, categories).
- Keep array size reasonable for better performance.
- Use indexes on array fields if frequently queried.
- Use **$addToSet** to avoid duplicate values.

Arrays make your data model flexible and powerful. Use them wisely for better data organization and faster queries!

MongoDB ARRAY QUERY OPERATORS

Find • Filter • Match • Work with Arrays

WHAT IS ARRAY QUERY?

Array query operators allow you to filter and search documents based on array values in MongoDB. They help you find documents that match specific elements inside an array.

KEY OPERATORS

1. **$all** – match all values
2. **$elemMatch** – match element with multiple conditions
3. **$size** – match array length

ARRAY DATA TYPES

```
{
  tags: [ "web", "dev", "js" ],
   scores: [85, 90, 78],
   users: [
      { name: "John", age: 25 },
      { name: "Bob", age: 30 }
   ]
}
```

Arrays can contain:
- values (strings, numbers, etc.)
- objects (nested documents)
- other arrays (nested arrays)

PRO TIP

Use the right operator for better performance.

- Create indexes on array fields if you query them often.
- Test your queries with small datasets first.

1. $ALL – MATCH ALL VALUES IN ARRAY

Find documents where the array contains all the specified values.

```
db.products.find({
  tags: { $all: ["mongodb", "python"] }
})
```

COLLECTION DATA

_id	name	tags
1	Mongo Book	["mongodb","python"]
2	Web Course	["javascript","node"]
3	Dev Tools	["mongodb","node","git"]

RESULT

Matches document _id: 1

```
{
  _id: 1,
  name: "Mongo Book",
  tags: ["mongodb", "python"]
}
```

2. $SIZE – MATCH ARRAY LENGTH

Find documents where the array has a specific number of elements.

```
db.users.find({
  skills: { $size: 3 }
})
```

RESULT

Returns documents where the skills array has exactly 3 elements.

3. $ELEMMATCH – MATCH ELEMENTS WITH CONDITIONS

Find documents where at least one element in the array matches multiple conditions.

```
db.users.find({
  scores: { $elemMatch: {
     $gt: 70,
     $lt: 90
  },
})
```

COLLECTION DATA

name	scores
Alice	[85, 91, 70]
Bob	[60, 75, 82]
Charlie	[72, 88, 90]

RESULT

Returns document:

```
{ name: "Bob",
  scores: [60, 75, 82] }
```

EXAMPLE: FINDING NESTED ARRAY DATA

$elemMatch works great with arrays of objects (nested data).

```
db.users.find({
  orders: { $elemMatch: {
     product: "Laptop",
     status: "Delivered",
  },
})
```

RESULT

Returns users who have a Laptop order with status "Delivered".

4. $ALL WITH OBJECTS – MULTIPLE CONDITIONS

Works with array of objects when you want to match multiple fields.

```
db.inventory.find({
  items: { $all: [
    | { name: "Apple", qty: 10 },
    | { name: "Banana", qty: 5 }
  ]
})
```

RESULT

Returns documents where the items array contains both Apple with qty 10 and Banana with qty 5.

5. $SIZE + $ALL TOGETHER (ADVANCED)

Use multiple operators together for powerful queries.

```
db.posts.find({
  tags: { $all: ["mongodb", "tutorial", "dev"] },
  comments: { $size: 3 }
})
```

RESULT

Docs with exactly 3 comments and tags including all the specified values.

REAL-WORLD USE CASES

Match tags in blog posts
Find posts with multiple tags (web + dev).

Filter search results
Get products with specific features (e.g., color + size).

Analytics / Reports
Find users with exact number of activities.

Match nested data
Find orders with specific product and status.

QUICK SUMMARY

Operator	Name	Use Case
$all	Match all values	Array contains all specified values
$elemMatch	Match element (multi conditions)	Element matches multiple conditions
$size	Match array length	Find arrays with specific number of elements

KEY POINTS

- $all requires all specified values (not just one).
- $elemMatch ensures a single array element matches all given conditions.
- $size matches exact array length.
- Can work with simple values, numbers, strings, objects and nested arrays.

REMEMBER

- Arrays are powerful for flexible data storage.
- Use correct operator to get accurate results.
- Better query → better performance.
- Always test queries before using in production.
- Combine operators for complex searches.

MongoDB ELEMENT OPERATORS

Query Documents Based on Field Existence and Data Type

WHAT ARE ELEMENT OPERATORS?

Element operators help you query documents based on the existence of a field or the data type of a field's value.

KEY OPERATORS

$exists — Matches documents that contain or do not contain a specified field.

$type — Matches documents where the field value is of a specific BSON data type.

QUICK FACTS

- Field names are case-sensitive.
- $exists checks field presence (even if value is null).
- $type checks the BSON data type of the field value.
- Useful for data validation and finding missing fields.

PRO TIP

Use $exists to find missing fields and keep your data consistent.
Use $type to filter data by exact data type.

SAMPLE DOCUMENTS (COLLECTION: users)

```
[
  { _id: 1, name: "Alice", age: 25, email: "alice@example.com", status: "active" },
  { _id: 2, name: "Bob", age: null, email: "bob@example.com" },
  { _id: 3, name: "Charlie", status: "inactive"},
  { _id: 4, name: "David", age: 30, email: null, score: 88 },
  { _id: 5, name: "Eva", age: "25", email: "eva@example.com" }
]
```

ELEMENT OPERATORS OVERVIEW

Operator	Purpose	Matches	Example Use
$exists	Matches documents that contain (true) or do not contain (false) the specified field.	• Field exists (true) • Field does not exist (false)	Find documents with or without a field.
$type	Matches documents where the field value is of a specific BSON data type.	• String, Number, Array, Object, Bool, Date, Null, etc.	Find documents where a field is of a specific data type.

BSON (Binary JSON) data types are used internally in MongoDB.

1. $EXISTS – MATCH FIELD PRESENCE

Field Exists (true)

Find documents where "email" field exists.

```
db.users.find({ email: { $exists: true } })
```

Result

_id	name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
4	David	null
5	Eva	eva@example.com

Includes documents where email is null.

Field Does Not Exist (false)

Find documents where "score" field does NOT exist.

```
db.users.find({ score: { $exists: false } })
```

Result

_id	name	score
1	Alice	—
2	Bob	—
3	Charlie	—
5	Eva	—

Only documents missing the field are returned.

2. $TYPE – MATCH DATA TYPE

Find documents where "age" is of type Number (Int32).

```
db.users.find({ age: { $type: "int" } })
```

Result

_id	name	age (type)
1	Alice	25 (Int32)
4	David	30 (Int32)

Other examples:

```
db.users.find({ age: { $type: "string" } })      // age stored as string (Eva)
db.users.find({ age: { $type: "null" } })        // age is null (Bob)
db.users.find({ status: { $type: "string" } })   // status is string
db.users.find({ email: { $type: "null" } })      // email is null (David)
```

BSON DATA TYPES (FOR $TYPE)

Type Name	Alias (in $type)	Example Value
Double	"double"	25.5
String	"string"	"Hello"
Object	"object"	{ name: "A" }
Array	"array"	[1, 2, 3]
Binary	"binData"	BinData(0, "....")
Undefined	"undefined"	(deprecated)
ObjectId	"objectId"	ObjectId("...")
Boolean	"bool"	true / false
Date	"date"	ISODate("2024-01-01")
Null	"null"	null
Regular Expression	"regex"	/abc/i
JavaScript	"javascript"	function() { }
Int32	"int"	25
Timestamp	"timestamp"	Timestamp(123, 1)
Int64	"long"	NumberLong(123456)
Decimal128	"decimal"	NumberDecimal("10.5")

COMMON USE CASES

- Find documents missing important fields.
- Validate data types in a collection.
- Clean up inconsistent or incomplete data.
- Build dynamic filters in applications.

EXAMPLE: COMBINE OPERATORS

Find documents where email exists AND is a string.

```
db.users.find({
  email: { $exists: true, $type: "string" }
})
```

Powerful filters for clean and reliable data.

MISSING FIELD vs NULL FIELD

Missing Field (Field does not exist)

```
{ name: "Charlie" }
```

VS

Null Field (Field exists with null value)

```
{ name: "David", email: null }
```

Both are different and handled differently in queries!

BEST PRACTICES

- Use $exists to find missing or optional fields.
- Use $type to ensure correct data types.
- Combine with other operators for advanced queries.
- Always test queries with real data.

Element operators ($exists, $type) help you build smart queries, find missing data, and maintain a healthy database.

MongoDB REGULAR EXPRESSIONS

Powerful Pattern Matching for Text Search

WHAT ARE REGULAR EXPRESSIONS?

Regular Expressions allow you to search for text patterns in strings. MongoDB provides $regex and $regexMatch for powerful text matching.

KEY OPERATORS

$regex — Query operator used in find() to match string patterns.

$regexMatch — Aggregation operator used in $project, $match, etc. to match patterns.

QUICK FACTS

- $regex is used in queries (find).
- $regexMatch is used in aggregation.
- Use 'i' option for case-insensitive search.
- Regular expressions are case-sensitive by default.
- Works only on string fields.

PRO TIP

Always use ^ and $ anchors for exact matches when possible. Use case-insensitive option (i) for better user experience.

SAMPLE DOCUMENTS (COLLECTION: users)

```
[
  { _id: 1, name: "Alice Johnson", email: "alice@example.com", city: "New York" },
  { _id: 2, name: "Bob Smith", email: "bobsmith@mail.com", city: "Boston" },
  { _id: 3, name: "Charlie Brown", email: "charlie.brown@domain.com", city: "Chicago" },
  { _id: 4, name: "David Wilson", email: "david.wilson@example.com", city: "Dallas" },
  { _id: 5, name: "Eve Adams", email: "eve_adams@sample.com", city: "Denver" },
  { _id: 6, name: "ALICE Cooper", email: "alice.cooper@music.com", city: "Los Angeles" }
]
```

1. $REGEX – FIND() QUERY EXAMPLES

Search Purpose	Query	Description	Matched Documents
Starts with 'Ali'	`db.users.find({ name: { $regex: /^Ali/ } })`	Names starting with Ali	Alice Johnson, ALICE Cooper
Contains 'son'	`db.users.find({ name: { $regex: /son/ } })`	Names containing son	Alice Johnson, David Wilson
Ends with 'own'	`db.users.find({ name: { $regex: /own$/ } })`	Names ending with own	Charlie Brown
Case insensitive search	`db.users.find({ name: { $regex: /alice/, $options: "i" } })`	Match 'alice' in any case (i option)	Alice Johnson, ALICE Cooper
Email contains 'example.com'	`db.users.find({ email: { $regex: /example\\.com$/i } })`	Emails ending with example.com (case-insensitive)	Alice Johnson, David Wilson

2. $REGEXMATCH – AGGREGATION EXAMPLES

```
db.users.aggregate([
  { $project: {
    name: 1,
    email: 1,
    hasExample: { $regexMatch: { input: "$email", regex: /example\.com$/i } }
  }}
])
```

Search Purpose	Aggregation Expression	Description	Result Field
Email ends with example.com	`{ $regexMatch: { input: "$email", regex: /example\\.com$/i } }`	Check if email ends with example.com (case-insensitive)	hasExample
Name contains 'brown'	`{ $regexMatch: { input: "$name", regex: /brown/i } }`	Check if name contains brown (any case)	hasBrown
Starts with 'E'	`{ $regexMatch: { input: "$name", regex: /^E/ } }`	Check if name starts with E	startsWithE

REGEX OPTIONS

- **i** — Case-insensitive: Match without considering case.
- **m** — Multi-line: ^ and $ match the start/end of each line.
- **s** — Dot matches all: Makes . match newline characters too.
- **x** — Extended mode: Allows whitespace and comments in pattern.
- **u** — Unicode: Enables full Unicode support.

COMMON PATTERNS

Pattern	Meaning	Example Match
^text	Starts with 'text'	^Ali → Alice, Alice Johnson
text$	Ends with 'text'	son$ → Johnson, Wilson
text	Contains 'text' (anywhere)	ice → Alice, Alice Cooper
^text$	Exact match	^Boston$ → Boston
.	Any single character	a.c → abc, axc
.*	Any number of characters	A.*e → Alice, Eve, Alice Cooper
[abc]	Any one character in set	[ae]ve → eve, ave
[^abc]	Any character NOT in set	[^A]lice → alice

PRACTICAL EXAMPLES

Use Case	Query	What it Finds
Find users with email from example.com	`db.users.find({ email: { $regex: /@example\\.com$/i } })`	All users whose email domain is example.com
Find names with exactly 3 letters	`db.users.find({ name: { $regex: /^[A-Za-z]{3}$/ } })`	Names having exactly 3 alphabetic characters
Find emails with gmail or yahoo	`db.users.find({ email: { $regex: /@@gmail\|yahoo)\.com$/i } })`	Emails from gmail.com or yahoo.com
Find city names starting with 'D'	`db.users.find({ city: { $regex: /^D/i } })`	Cities starting with letter D or d

BEST PRACTICES

- Use $regex with indexes carefully for better performance.
- Anchor your pattern (^, $) when possible to speed up search.
- Use $options: "i" for case-insensitive search.
- Avoid leading wildcards (like /re/) to utilize indexes.
- Test your regex patterns at regex101.com before using.

Regular Expressions in MongoDB make text search flexible and powerful. Use $regex in queries and $regexMatch in aggregation for advanced text filtering!

Search Smarter. Filter Better.

MongoDB AGGREGATION BASICS

Powerful Data Analysis with Pipeline Stages

Transform
Group
Filter
Summarize

WHAT IS AGGREGATION?

Aggregation in MongoDB is used to analyze and process data. It helps you transform multiple documents into meaningful results using a pipeline of stages.

KEY TERMS

Aggregation
Process data using pipeline.

Pipeline
An array of stages.

Stages
Individual operations like $match, $group, etc.

Documents
Input and output data.

Transform Data
Reshape, filter and summarize.

SAMPLE COLLECTION (students)

```
[
 { "_id": 1, "name": "Alice", "age": 25, "city": "Delhi", "marks": 78 },
 { "_id": 2, "name": "Bob", "age": 22, "city": "Mumbai", "marks": 85 },
 { "_id": 3, "name": "Charlie", "age": 24, "city": "Delhi", "marks": 92 },
 { "_id": 4, "name": "David", "age": 23, "city": "Bangalore", "marks": 70 },
 { "_id": 5, "name": "Eva", "age": 26, "city": "Mumbai", "marks": 88 },
 { "_id": 6, "name": "Frank", "age": 21, "city": "Delhi", "marks": 95 },
 { "_id": 7, "name": "Grace", "age": 24, "city": "Chennai", "marks": 80 }
]
```

1. AGGREGATION SYNTAX

```
db.collection.aggregate([
   { $match: { <condition> } },      // Stage 1
   { $group:   { <group spec> } },   // Stage 2
   { $project: { <field spec> } },   // Stage 3
   { $sort:    { <sort spec> } },    // Stage 4
   { $limit:   { <number> } },       // Stage 5
   { $skip:    { <number> } }        // Optional
])
```

A pipeline is an array of stages.
Each stage does a specific task.

2. COMMON AGGREGATION STAGES

Stage	Purpose
$match	Filter documents
$group	Group documents and calculate results
$project	Select or reshape fields
$sort	Sort the results
$limit	Limit the number of results
$skip	Skip a number of results
$unwind	Deconstruct array fields
$lookup	Join with another collection

3. EXAMPLE: 1. FILTER DATA ($match)

Find all students from Delhi

```
db.students.aggregate([
  { $match: { city: "Delhi" } }
])
```

Result

_id	name	city	age	marks
1	Alice	Delhi	25	78
3	Charlie	Delhi	24	92
6	Frank	Delhi	21	95

4. EXAMPLE: 2. GROUP DATA ($group)

Get total marks per city

```
db.students.aggregate([
  { $group: { _id:
    "$totalMarks: { $sum: "$marks"
])
```

_id	totalMarks
Delhi	225
Mumbai	173
Bangalore	70
Chennai	80

5. EXAMPLE: 3. PROJECT FIELDS ($project)

Show only name and marks

```
db.students.aggregate([
  { $project: { _id:
    _id: 0, name: 1, marks: 1 } }
])
```

Result

name	marks
Alice	78
Bob	85
Charlie	92
David	70
Eva	88
Frank	95
Grace	80

6. EXAMPLE: 4. GROUP + PROJECT

Get number of students per city

```
db.students.aggregate([
  { $group: {
  { _id: "$city", count: { $sum: 1 } }},
  { $project: { city: "$_id",
    totalStudents: "$count", _id: 0 } }
])
```

city	totalStudents
Delhi	3
Mumbai	2
Bangalore	1
Chennai	1

7. EXAMPLE: 5. SORT + LIMIT

Top 3 students by marks

```
db.students.aggregate([
  { $sort: { marks: -1 } },
  { $limit: 3 },
  { $project: { _id: 0, name: 1,
    marks: 1 } }
])
```

name	marks
Frank	95
Charlie	92
Eva	88

8. EXAMPLE: 6. UNWIND (Arrays)

Unwind tags and count each tag

```
db.products.aggregate([
  { $unwind: "$tags" },
  { $group: { _id: "$tags",
    total: { $sum: 1 } } }
])
```

Result

tag	total
laptop	3
gaming	2
education	2
office	1

MORE USEFUL EXAMPLES

1. Complex Pipeline
Combine multiple stages for advanced analysis.

```
db.sales.aggregate([
  { $match: { status: "paid" } },
  { $group: { _id: "$product",
      total: { $sum: "$amount" } },
    { $sort: { total: -1 } }
])
```

2. Match Array Elements
Find documents with specific elements in an array.

```
db.students.find({
   scores: { $elemMatch: { $gte: 90 } }
})
```

3. Multiple Groups
Group by city and age.

```
db.students.aggregate([
  { $group: {
    _id: { city: "$city", age: "$age" },
    count: { $sum: 1 }
  } }
])
```

4. Final Results
Limit and format output.

```
db.students.aggregate([
  { $match: { city: "Mumbai" } },
  { $group: { _id: null,
    avgMarks: { $avg: "$marks" } },
  { $project: { avgMarks: 1, _id: 0 }
])
```

KEY TAKEAWAYS

- Aggregation helps you analyze and transform data.
- Use pipeline stages to filter, group, calculate and sort.
- Stages can be combined for powerful results.
- Very useful for reports, analytics and dashboards.
- Play with different stages to get the best results.

PRO TIP

Start with $match to reduce data, then use $group, $project and $sort for better performance and clean results.

Good data analysis leads to better decisions!

mongoDB.
Query • Analyze • Build • Grow

Same Database | A Smarter Way | MongoDB | Learn • Practice • Build Real Projects • Grow Your Skills | Make Data Work for You

MongoDB $match

Filter Documents Early in the Aggregation Pipeline

WHAT IS $MATCH?

`$match` is an aggregation stage used to filter documents based on specified conditions. It works like the `find()` query and should be placed early in the pipeline for best performance.

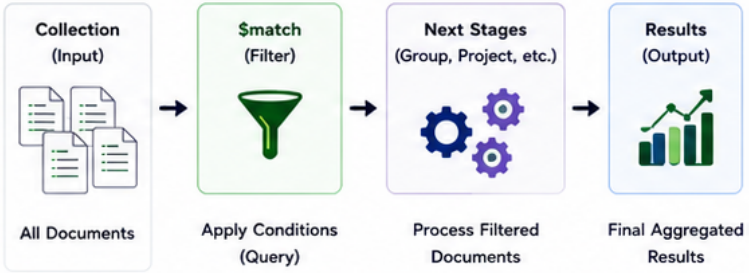
KEY POINTS

- Filters documents based on conditions.
- Uses the same query operators as find().
- Place `$match` as the first stage for better performance.
- Reduces the number of documents passed to next stages.

PRO TIP

Always use `$match` as the first stage in your pipeline to reduce the data size early and improve speed.

PIPELINE FLOW WITH $MATCH



Use `$match` early → Reduce Data → Improve Performance

EXAMPLE: BEFORE AND AFTER $MATCH

Before $match (All Documents)

_id	name	category	price	qty
1	Laptop	Electronics	1200	5
2	Phone	Electronics	800	10
3	T-Shirt	Clothing	20	15
4	Jeans	Clothing	40	8
5	Headphones	Electronics	200	12
6	Book	Books	15	30

$match

After $match (Filtered Results)

_id	name	category	price	qty
1	Laptop	Electronics	1200	5
2	Phone	Electronics	800	10
5	Headphones	Electronics	200	12

Filter Condition

```
{ category: "Electronics", price: { $gte: 200 } }
```

Only documents that match the condition will pass to the next stages.

SYNTAX

```
db.collection.aggregate([
   { $match: { <query> } },
   { <next stage> },
   ...
])
```

EXAMPLE QUERY

```
db.products.aggregate([
   { $match: { category: "Electronics", price: { $gte: 200 } } },
   { $group: { _id: "$category", totalSales: { $sum: "$price" } } },
   { $project: { _id: 0, category: "$_id", totalSales: 1 } }
])
```

COMMON QUERY OPERATORS IN $MATCH

Operator	Meaning	Operator	Meaning
$eq	Equal	$lte	Less than or equal
$ne	Not equal	$in	In an array
$gt	Greater than	$nin	Not in an array
$gte	Greater than or equal	$and	Logical AND
$lt	Less than	$or	Logical OR

EXAMPLES

1. Match Single Condition

```
{ $match: { category: "Electronics" } }

// Matches documents where category
// is "Electronics"
```

2. Match Multiple Conditions

```
{ $match: { category: "Electronics",
            price: { $gte: 200 },
            qty: { $gt: 5 } } }

// Matches Electronics with price >= 200
// and quantity > 5
```

3. Using $or Operator

```
{ $match: { $or: [
   { category: "Electronics" },
   { price: { $gte: 1000 } }
] } }

// Matches documents in Electronics
// OR price >= 1000
```

WHY USE $MATCH EARLY?

- **Better Performance** — Reduces the number of documents processed in later stages.
- **Less Data Transfer** — Sends fewer documents between pipeline stages.
- **Optimized Memory Usage** — Uses less memory and system resources.
- **Faster Results** — Returns results quickly by working on smaller dataset.

`$match` is your first step to smart aggregation. Filter early, process less, and get better performance!

MongoDB $group

Group Data by Fields and Perform Aggregations

WHAT IS $GROUP?

$group is an aggregation stage used to group documents by a specified key (_id) and perform calculations on each group. It is typically used after $match to summarize data.

KEY CONCEPTS

- **_id** — The field or expression used to group documents.
- **Accumulator Operators** — Perform calculations on each group.
- **Group Results** — Returns one document per group with calculated values.

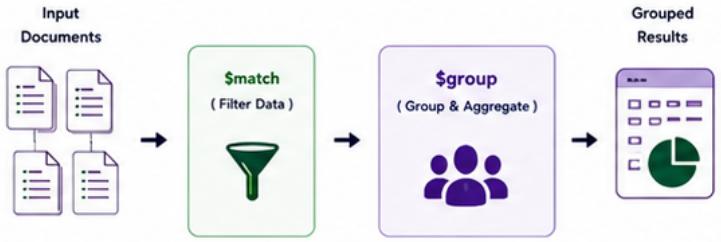
QUICK FACTS

- $group reduces multiple documents into a single summary document.
- _id can be a field, multiple fields, or an expression.
- Use accumulator operators to calculate values for each group.
- $group is memory intensive; use $match early to reduce data.

PRO TIP

Always use $match before $group to reduce the number of documents processed and improve performance.

AGGREGATION PIPELINE WITH $GROUP



$group groups documents by the _id expression and calculates aggregated values.

SAMPLE DATA (COLLECTION: sales)

_id	category	item	price	qty
1	Electronics	Laptop	1200	1
2	Electronics	Phone	800	2
3	Electronics	Tablet	400	3
4	Clothing	T-Shirt	20	5
5	Clothing	Jeans	40	2
6	Books	Book	15	4
7	Books	Notebook	10	6

$GROUP OUTPUT EXAMPLE

_id (category)	totalCount	totalQty	totalSales	avgPrice	minPrice	maxPrice
Electronics	3	6	2400	800	400	1200
Clothing	2	7	100	30	20	40
Books	2	10	25	12.5	10	15

EQUIVALENT SQL

```sql
SELECT category,
       COUNT(*) AS totalCount,
       SUM(qty)  AS totalQty,
       SUM(price*qty) AS totalSales,
       AVG(price)  AS avgPrice,
       MIN(price)  AS minPrice,
       MAX(price)  AS maxPrice
FROM sales
GROUP BY category;
```

$GROUP SYNTAX

```
{
  $group: {
    _id: <expression>,
    <field1>: { <accumulator1>: <expression> },
    <field2>: { <accumulator2>: <expression> },
    ...
  }
}
```

EXAMPLE QUERY

```
db.sales.aggregate([
    { $match: { price: { $gt: 0 } } },
    { $group: {
        _id: "$category",
        totalCount: { $count: {} },
        totalQty:   { $sum: "$qty" },
        totalSales: { $sum: { $multiply: ["$price", "$qty"] } },
        avgPrice:   { $avg: "$price" },
        minPrice:   { $min: "$price" },
        maxPrice:   { $max: "$price" }
    }}
])
```

COMMON ACCUMULATOR OPERATORS IN $GROUP

Operator	Description	Example	Meaning
$count	Counts number of documents	{ $count: {} }	Total documents in group
$sum	Calculates sum of values	{ $sum: "$qty" }	Sum of qty
$avg	Calculates average	{ $avg: "$price" }	Average price
$min	Finds minimum value	{ $min: "$price" }	Minimum price
$max	Finds maximum value	{ $max: "$price" }	Maximum price
$first	Returns first value	{ $first: "$item" }	First item in group
$last	Returns last value	{ $last: "$item" }	Last item in group
$push	Adds values to an array	{ $push: "$item" }	Array of all items
$addToSet	Adds unique values to array	{ $addToSet: "$item" }	Unique items array

PRACTICAL EXAMPLES

1. Count Documents by Category

```
{ $group: {
    _id: "$category",
    totalCount: { $count: {} }
}}
```

Result: { _id: "Electronics", totalCount: 3 }

2. Total Sales by Category

```
{ $group: {
    _id: "$category",
    totalSales: { $sum: {
    $multiply: ["$price", "$qty"] } }
}}
```

Result: { _id: "Electronics", totalSales: 2400 }

3. Average Price by Category

```
{ $group: {
    _id: "$category",
    avgPrice: { $avg: "$price"
}}
```

Result: { _id: "Electronics", avgPrice: 800 }

4. Min & Max Price by Category

```
{ $group: {
    _id: "$category",
    minPrice: { $min: "$price"},
    maxPrice: { $max: "$price" }
}}
```

Result: { _id: "Books", minPrice: 10, maxPrice: 15 }

5. Collect Items into Array

```
{ $group: {
    _id: "$category",
    items: { $push: "$item" }
}}
```

Result: {_id: "Books", items: ["Book", "Notebook"] }

BEST PRACTICES

- Use $match before $group to filter early.
- Group on indexed fields when possible.
- Avoid high cardinality in _id.
- Monitor memory usage for large datasets.
- Use allowDiskUse(true) for large aggregations.

$group helps you summarize and analyze your data. Choose the right accumulator operators to get powerful insights!

MongoDB $project

Reshape Your Documents – Include, Exclude, Rename & Calculate Fields

WHAT IS $PROJECT?

`$project` is an aggregation stage used to reshape each document in the pipeline. You can include fields, exclude fields, rename fields, or create calculated fields.

KEY CONCEPTS

- **Include Fields (1)** — Show only the fields you need.
- **Exclude Fields (0)** — Hide fields you don't need. (`_id` is included by default)
- **Rename Field** — Change the field name in the output.
- **Calculated Field** — Create new fields using expressions.

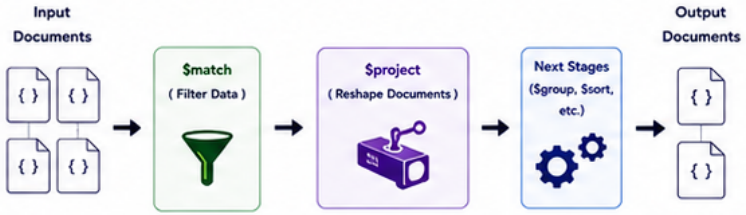
QUICK FACTS

- 1 = Include a field in output.
- 0 = Exclude a field from output.
- Cannot mix 1 and 0 (except _id).
- Fields are reshaped in the output.
- Useful for reducing data size and improving performance.

PRO TIP

Use $project as early as possible in the pipeline to reduce the amount of data passed to the next stages.

AGGREGATION PIPELINE WITH $PROJECT



`$project` decides what each output document looks like.

EXAMPLE: INPUT AND OUTPUT

Input Documents (Collection: users)

_id	name	age	email	city	salary
1	Alice	25	alice@mail.com	Delhi	50000
2	Bob	30	bob@mail.com	Mumbai	60000
3	Charlie	28	charlie@mail.com	Pune	55000

$project Example (Include name & city)

```
{ $project: {
    _id: 0,
    name: 1,
    city: 1
}}
```

Output Documents

name	city
Alice	Delhi
Bob	Mumbai
Charlie	Pune

`_id: 0` → Excludes the _id field from output.

1. INCLUDE FIELDS (1)

Show only the selected fields.

```
{ $project: {
    _id: 0,
    name: 1,
    email: 1
}}
```

Output

name	email
Alice	alice@mail.com
Bob	bob@mail.com
Charlie	charlie@mail.com

2. EXCLUDE FIELDS (0)

Exclude the selected fields.

```
{ $project: {
    age: 0,
    salary: 0
}}
```

Output

_id	name	email	city
1	Alice	alice@mail.com	Delhi
2	Bob	bob@mail.com	Mumbai
3	Charlie	charlie@mail.com	Pune

3. RENAME FIELD

Rename a field using a new name.

```
{ $project: {
    _id: 0,
    fullName: "$name",
    location: "$city"
}}
```

Output

fullName	location
Alice	Delhi
Bob	Mumbai
Charlie	Pune

4. CALCULATED FIELD

Create new fields using expressions.

```
{ $project: {
    _id: 0,
    name: 1,
    monthlySalary: { $divide: [ "$salary", 12 ] },
    isHighSalary: { $gt: [ "$salary", 55000 ] }
}}
```

Output

name	monthlySalary	isHighSalary
Alice	4166.67	false
Bob	5000.00	true
Charlie	4583.33	false

$PROJECT SYNTAX

```
{ $project: {
    field1: <1 | 0 | expression>,
    field2: <1 | 0 | expression>,
    ...
    _id: <0 | 1>  // optional
}}
```

- `1` → Include field
- `0` → Exclude field
- `Expression` → Calculate or rename field

MORE EXAMPLES

Keep _id only

```
{ $project: { name: 0, age: 0,
  email: 0, city: 0, salary: 0 }}
```

Output

_id
1
2
3

Add Full Name

```
{ $project: { _id: 0,
  fullName: { $concat: [ "$name", " ", "$city" ] }}
```

Output

fullName	
Alice	Delhi
Bob	Mumbai
Charlie	Pune

Conditional Field

```
{ $project: { name: 1,
  bonus: { $cond: [ { $gt: [ "$salary", 55000 ] },
  "High", "Low" ] }
```

Output

name	bonus
Alice	Low
Bob	High
Charlie	Low

BEST PRACTICES

- Use $project early to reduce data size.
- Include only fields you really need.
- Exclude large or unused fields.
- Rename fields for better readability.
- Create calculated fields for insights.
- Always exclude _id if not required using `_id: 0`.

`$project` helps you control the shape of your data. Use it to include, exclude, rename and calculate fields for clean and efficient output!

MongoDB $sort

Sort Your Documents – Ascending (1) or Descending (-1)

WHAT IS $SORT?

$sort is an aggregation stage used to sort documents by one or more fields. You can sort in Ascending (1) or Descending (-1) order.

KEY POINTS

- 1 = Ascending (A to Z, 0 to 9)
- -1 = Descending (Z to A, 9 to 0)
- Sort by one field or multiple fields.
- Use $sort after $match and before $limit for best performance.

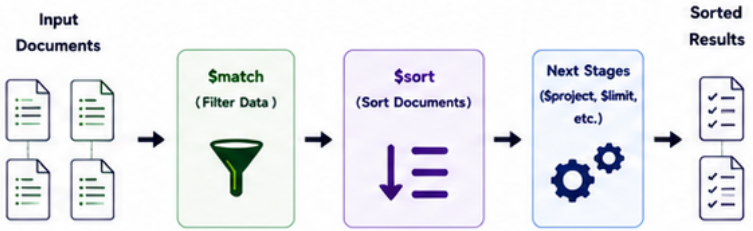
QUICK REFERENCE

Value	Order
1	Ascending (A → Z, 0 → 9)
-1	Descending (Z → A, 9 → 0)

PRO TIP

Always $match first, then $sort, then $limit to get fast and optimized results.

AGGREGATION PIPELINE WITH $SORT



$sort orders the documents based on the specified field(s) and the order (1 or -1).

EXAMPLE: SORT BY SINGLE FIELD

Input Documents (Collection: products)

_id	name	category	price	rating
1	Laptop	Electronics	1200	4.5
2	Smartphone	Electronics	800	4.2
3	Headphones	Electronics	150	4.6
4	T-Shirt	Clothing	20	4.0
5	Jeans	Clothing	40	4.3

$sort { price: 1 } (Ascending) →

Sorted Results (By price ASC)

_id	name	price
4	T-Shirt	20
5	Jeans	40
3	Headphones	150
2	Smartphone	800
1	Laptop	1200

1 = Ascending | -1 = Descending

1. SORT BY SINGLE FIELD (ASC)

Sort by price in Ascending order

```
db.products.aggregate([
   { $sort: { price: 1 } }
])
```

Output

_id	name	price
4	T-Shirt	20
5	Jeans	40
3	Headphones	150
2	Smartphone	800
1	Laptop	1200

ASC ↑

2. SORT BY SINGLE FIELD (DESC)

Sort by rating in Descending order

```
db.products.aggregate([
   { $sort: { rating: -1 } }
])
```

Output

_id	name	rating
3	Headphones	4.6
1	Laptop	4.5
5	Jeans	4.3
2	Smartphone	4.2
4	T-Shirt	4.0

DESC ↓

3. SORT BY MULTIPLE FIELDS

Sort by category ASC, then price DESC

```
db.products.aggregate([
   { $sort: { category: 1, price: -1 } }
])
```

Output

_id	name	category	price
5	Jeans	Clothing	40
4	T-Shirt	Clothing	20
1	Laptop	Electronics	1200
2	Smartphone	Electronics	800
3	Headphones	Electronics	150

COMMON USE CASES

- Sort products by price (Low to High / High to Low)
- Sort students by marks (Topper list)
- Sort by date (Newest first)
- Sort by multiple fields (category, then price)
- Use with $limit to get Top N results

$SORT SYNTAX

```
{ $sort: { field1: 1 | -1, field2: 1 | -1, ... } }
```

- 1 → Ascending order
- -1 → Descending order

MORE EXAMPLES

Sort by name ASC

```
{ $sort: { name: 1 } }
```

(A → Z)

A → Z

Sort by price DESC

```
{ $sort: { price: -1 } }
```

(High → Low)

(High → Low)

Sort by rating DESC, then name ASC

```
{ $sort: { rating: -1, name: 1 } }
```

Z → ↑ , ↑ → ?

Z → A A → Z

BEST PRACTICES

- Use $match before $sort to reduce data size.
- Create indexes on fields used in $sort.
- Use $limit after $sort to get top results.
- Sorting large datasets without index can be slow.

Use $sort to organize your data in the order you need – either 1 (ASC) or -1 (DESC) – single or multiple fields!

MongoDB $limit & $skip

Pagination with $limit & $skip – Get the Results You Need

WHAT ARE $LIMIT & $SKIP?

$limit
Limits the number of documents passed to the next stage.
Returns a specific number of documents.

$skip
Skips the specified number of documents and passes the remaining documents.

KEY POINTS

- $skip ignores the first N documents.
- $limit returns only N documents.
- Use together for Pagination.
- Always use $sort before $skip for consistent results.

PAGINATION FORMULA

Page Number (p) starts from 1

$skip = (p - 1) * Page Size

$limit = Page Size

PRO TIP

Always use $sort before $skip and $limit to get stable and predictable results.

AGGREGATION PIPELINE WITH $LIMIT & $SKIP

$skip selects where to start, $limit decides how many documents to return.

EXAMPLE: PAGINATION (Collection: users)

All Documents (Sorted by name ASC)

_id	name	age	city
1	Alice	25	Delhi
2	Bob	28	Mumbai
3	Charlie	22	Pune
4	David	30	Delhi
5	Eve	24	Noida
6	Frank	27	Bangalore
7	Grace	26	Chennai
8	Henry	29	Kolkata

Page Size = 3

Page 1 (p = 1) — $skip: 0, $limit: 3

_id	name	age
1	Alice	25
2	Bob	28
3	Charlie	22

Page 2 (p = 2) — $skip: 3, $limit: 3

_id	name	age
4	David	30
5	Eve	24
6	Frank	27

Page 3 (p = 3) — $skip: 6, $limit: 3

_id	name	age
7	Grace	26
8	Henry	29

How it works?

$skip
Skips the first N documents.

$limit
Limits the next N documents.

Result
Only limited documents are passed to the next stage.

$LIMIT SYNTAX

```
db.collection.aggregate([
  { $limit: <number> }
])
```

Returns first <number> documents.

$SKIP SYNTAX

```
db.collection.aggregate([
  { $skip: <number> }
])
```

Skips first <number> documents.

$LIMIT + $SKIP SYNTAX (PAGINATION)

```
db.collection.aggregate([
  { $sort: { name: 1 } },  // Always sort first
  { $skip: <skip_value> },
  { $limit: <limit_value> }
])
```

REAL EXAMPLE QUERY (Page 2, Size = 3)

```
db.users.aggregate([
  { $sort: { name: 1 } },
  { $skip: 3 },      // (2 - 1) * 3 = 3
  { $limit: 3 }
])
```

Result

_id	name	age
4	David	30
5	Eve	24
6	Frank	27

PAGINATION EXAMPLES (Page Size = 3)

Page	Formula	$skip	$limit	Documents Returned
1	(1 - 1) * 3	0	3	1, 2, 3
2	(2 - 1) * 3	3	3	4, 5, 6
3	(3 - 1) * 3	6	3	7, 8
4	(4 - 1) * 3	9	3	(No more documents)

COMPLETE PIPELINE EXAMPLE

```
db.users.aggregate([
  { $match: { city: "Delhi" } },  // Filter
  { $sort: { age: -1 },           // Sort by age desc
  { $skip: 2 },                   // Skip first 2 docs
  { $limit: 2 }                   // Return next 2 docs
])
```

Sample Output

_id	name	age	city
5	Eve	24	Delhi
1	Alice	25	Delhi

BEST PRACTICES

- Always use $sort before $skip for stable results.
- Use $limit to reduce data in next stages.
- Great for Pagination in APIs.
- Avoid large $skip values on big collections (performance impact).

Use $skip to move forward and $limit to control how many documents you want. Perfect for Pagination!

MongoDB $unwind

Deconstruct Array Fields – Flatten Arrays into Separate Documents

WHAT IS $UNWIND?

$unwind is an aggregation stage that deconstructs an array field from the input documents to output a document for each element in the array.

KEY CONCEPTS

- Deconstructs array field. Creates a new document for each array element.
- Flattens nested arrays. Converts array items into individual documents.
- Preserves other fields. Other fields remain the same in the output documents.
- Works in aggregation pipeline. Used to process and transform nested array data.

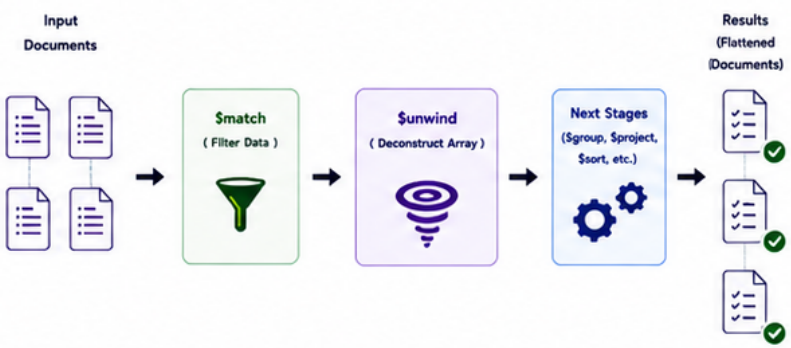
QUICK FACTS

- $unwind works only on array fields.
- One input document may generate multiple output documents.
- Useful for reporting, grouping, sorting on array elements.
- Use preserveNullAndEmptyArrays option to keep empty or missing arrays.

PRO TIP

Use $unwind early in the pipeline when you need to work with individual array elements in later stages like $match or $group.

AGGREGATION PIPELINE WITH $UNWIND



$unwind deconstructs an array field so that each array element becomes a separate document.

EXAMPLE: $UNWIND A SIMPLE ARRAY

Input Documents (Collection: students)

```
{
  {
    "_id": 1,
    "name": "Alice",
    "subjects": ["Math", "Science", "English"],
    "age": 20
  },
  {
    "_id": 2,
    "name": "Bob",
    "subjects": ["Math", "History"],
    "age": 21
  }
]
```

$unwind "$subjects"

Output Documents (After $unwind)

_id	name	subjects	age
1	Alice	Math	20
1	Alice	Science	20
1	Alice	English	20
2	Bob	Math	21
2	Bob	History	21

Each element of the "subjects" array becomes a new document.

$UNWIND SYNTAX (Basic)

```
db.collection.aggregate([
   { $unwind: "$arrayField" }
])

Example:
{ $unwind: "$subjects" }
// Deconstructs the subjects array
```

$UNWIND WITH OPTIONS

```
db.collection.aggregate([
  { $unwind: {
      path: "$arrayField",
      preserveNullAndEmptyArrays: true,
      includeArrayIndex: "index"
  } }
])
```

OPTIONS EXPLAINED

Option	Description
path	Required. The array field to deconstruct.
preserveNullAndEmptyArrays	If true, includes documents where the array field is null, missing or empty.
includeArrayIndex	Adds a new field with the index of each element in the array.

WITH preserveNullAndEmptyArrays: true

Input (subjects)	Output (preserveNullAndEmptyArrays: true)
["Math", "Science"]	2 documents
[] (empty array)	1 document (subjects: null)
null	1 document (subjects: null)
(field missing)	1 document (subjects: null)

$UNWIND WITH includeArrayIndex

```
db.students.aggregate([
  { $unwind: {
    path: "$subjects",
    includeArrayIndex: "idx"
  } }
])
```

Output (with index field "idx")

_id	name	subjects	idx	age
1	Alice	Math	0	20
2	Alice	Science	1	20
1	Alice	English	2	20
2	Bob	Math	0	21
2	Bob	History	1	21

$UNWIND NESTED ARRAYS

Input Document

```
{
  "_id": 1,
  "name": "John",
  "courses": [
    {"title": "DB", "tags": ["NoSQL", "MongoDB"] },
    {"title": "JS", "tags": ["Frontend", "ES6"] }
  ]
}
```

$unwind: "$couress"

$unwind: "$courses.tags"

Output Documents

_id	name	courses.title	courses.tags
1	John	DB	NoSQL
1	John	DB	MongoDB
1	John	JS	Frontend
2	John	JS	ES6

BEST PRACTICES

- Use $match before $unwind to reduce data early.
- Unwind only the array you need.
- Use preserveNullAndEmptyArrays: true when you don't want to lose documents.
- Use includeArrayIndex for ordering or debugging.
- Be careful: $unwind can increase the number of documents and affect performance.

$unwind is powerful for working with arrays. It converts array elements into documents so you can filter, group, or sort them easily!

MongoDB $lookup

Join Collections – Combine Related Data in a Single Query

WHAT IS $LOOKUP?

$lookup is an aggregation stage that performs a left outer join to another collection in the same database.

KEY CONCEPTS

- **Join Collections** — Combine data from two collections.
- **Local Collection** — The collection from which the aggregation starts.
- **Foreign Collection** — The collection to join data from.
- **Local Field** — Field in the local collection.
- **Foreign Field** — Field in the foreign collection that matches the local field.
- **as (Output Field)** — Name of the array field to store the joined data.

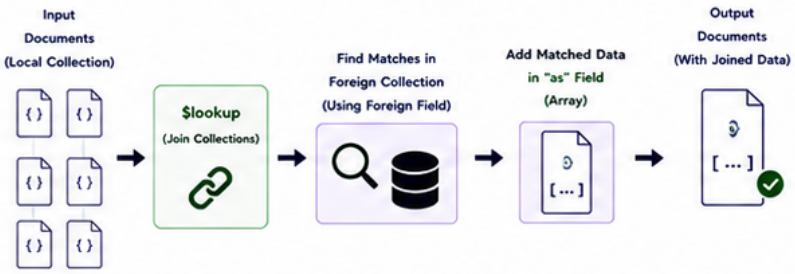
QUICK FACTS

- Performs Left Outer Join.
- Result is always an array in the "as" field.
- If no match found → empty array [].
- Can join on equality of fields.
- Works in a single database.

PRO TIP

Create indexes on Local Field and Foreign Field for better performance.

HOW $LOOKUP WORKS (PIPELINE FLOW)



$lookup adds an array field with matching documents from the foreign collection.

EXAMPLE: JOIN ORDERS WITH USERS

Local Collection: orders

_id	userId	item	amount
1	101	Laptop	1200
2	102	Mobile	800
3	101	Keyboard	200

Foreign Collection: users

_id	name	email	city
101	Alice	alice@mail.com	Delhi
102	Bob	bob@mail.com	Mumbai
103	Charlie	charlie@mail.com	Pune

$lookup Output (orders with user data)

_id	item	amount	user
1	Laptop	1200	[{"_id":101, "name":"Alice", "email":"alice@mail.com", "city":"Delhi"}]
2	Mobile	800	[{"_id":102, "name":"Bob", "email":"bob@mail.com", "city":"Mumbai"}]
3	Keyboard	200	[{"_id":101, "name":"Alice", "email":"alice@mail.com", "city":"Delhi"}]

Matches are returned as an array in the "as" field.
If no match → empty array [].

$LOOKUP SYNTAX (Basic)

```
{
  $lookup: {
    from: "foreignCollection",
    localField: "localFieldName",
    foreignField: "foreignFieldName",
    as: "outputArrayField"
  }
}
```

- **from:** Foreign collection to join.
- **localField:** Field from local collection.
- **foreignField:** Field from foreign collection.
- **as:** Name of the output array field.

$LOOKUP SYNTAX (With Pipeline)

```
{
  $lookup: {
    from: "foreignCollection",
    let: { localVar: "$localFieldName" },
    pipeline: [
      { $match: { $expr: { $eq: [ "$foreignFieldName",
                                  "$$localVar" ] } } },
      { $project: { name: 1, email: 1, city: 1 } }
    ],
    as: "outputArrayField"
  }
}
```

SYNTAX ELEMENTS EXPLAINED

Element	Description
from	Name of the foreign collection.
localField	Field from input documents.
foreignField	Field from foreign collection used for matching.
let (optional)	Define variables from local docs (used in pipeline).
pipeline (optional)	Advanced filtering on foreign data.
as	Output array field name.

WITH UNWIND (To Flatten Results)

```
{
  { $lookup: { from: "users",
               localField: "userId",
               foreignField: "_id",
               as: "user" } },
  { $unwind: "$user" }
]
```

Result:
"user" becomes a single document instead of an array.

EXAMPLE QUERY

```
{
  { $lookup: {
      from: "users",
      localField: "userId",
      foreignField: "_id",
      as: "userDetails"
    }
  }
]
```

orders → users

EDGE CASES

- **No Match Found:** Output array will be []
- **Multiple Matches:** All matching documents are returned in the array.
- **Different Field Names:** Use localField and foreignField.

PERFORMANCE TIPS

- Index foreignField in foreign collection.
- Project only required fields in pipeline.
- Use $match early in pipeline for better performance.
- Avoid $lookup on large unindexed fields.

BEST PRACTICES

- Use $lookup for simple equality joins.
- Use Pipeline version for complex conditions.
- Apply $unwind after $lookup if you need flat results.
- Always test with small datasets first.

$lookup is the power of Joins in MongoDB. Use it to combine related data and build powerful aggregations!

MongoDB $count

Count Documents in Aggregation Pipeline

WHAT IS $COUNT?

$count is an aggregation stage that counts the number of documents and returns the total count as a result.

KEY CONCEPTS

- Counts the number of documents passed from previous stage.
- Returns a single document with the total count.
- Useful for statistics and reporting.
- Always place at the end of the pipeline.

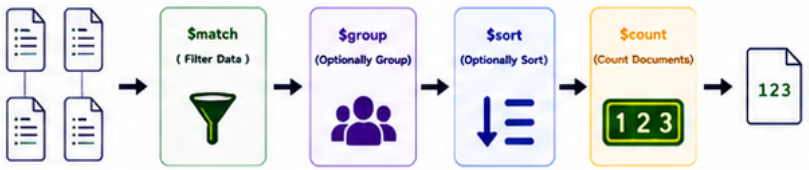
QUICK FACTS

- Returns a single document.
- Field name is required.
- If no documents, returns 0.
- Better performance than $group for simple counts.

PRO TIP

Use $count instead of $group with $sum: 1 for better performance and simplicity.

AGGREGATION PIPELINE WITH $COUNT



$count returns the total number of documents from the previous stage.

EXAMPLE: COUNT DOCUMENTS

Collection: products

_id	name	category	price
1	Laptop	Electronics	1200
2	Smartphone	Electronics	800
3	Headphones	Electronics	150
4	T-Shirt	Clothing	20
5	Jeans	Clothing	40
6	Watch	Accessories	1200

Count all documents

```
db.products.aggregate([
  { $count: "total" }
])
```

Result

total
6

Count documents with condition

```
db.products.aggregate([
  { $match: { category: "Electronics" } },
  { $count: "total" }
])
```

Result

total
3

$COUNT SYNTAX

```
{ $count: "<fieldName>" }
```

Field	Description
fieldName	Name of the field that will store the count.

$COUNT VS $GROUP

Using $count

```
db.collection.aggregate([
  { $match: { status: "A" } },
  { $count: "total" }
])
```

Simple, fast and optimized for counting.

Using $group

```
db.collection.aggregate([
  { $match: { status: "A" } },
  { $group: { _id: null,
    loo: total: { $sum: 1 } } }
])
```

More complex and slower for simple count.

$COUNT WITH OTHER STAGES

```
db.orders.aggregate([
  { $match: { status: "Delivered" } },
  { $group: { _id: "$customerId",
              totalOrders: { $sum: 1 } } },
  { $sort: { totalOrders: -1 } },
  { $count: "totalCustomers" }
])
```

Counts total customers after grouping and sorting.

EDGE CASES

- If no documents match the condition, returns a document with count 0.
- If pipeline is empty (no documents in collection), still returns 0.
- Field name in $count is required, otherwise error.
- $count must be the last stage in the pipeline.

EXAMPLE: REAL WORLD USE CASES

- Count total users in the system
- Count active users
- Count products in a category
- Count orders in a date range
- Count documents after filtering data

EXAMPLE OUTPUT

Pipeline

```
db.products.aggregate([
  { $match: { price: { $gt: 100 } },
  { $count: "total" }
])
```

Output

total
4

COMPLETE PIPELINE EXAMPLE

```
db.orders.aggregate([
  { $match: { status: "Delivered" } },
  { $group: { _id: "$city",
              totalOrders: { $sum: 1 } } },
  { $sort: { totalOrders: -1 } },
  { $count: "citiesCount" }
])
```

Output

citiesCount
12

BEST PRACTICES

- Use $count for simple counting.
- Place $count at the end of the pipeline.
- Use meaningful field name.
- Use $match early to reduce documents before counting.

$count is the fastest way to get the total number of documents in your aggregation result!

MongoDB Accumulator Operators

Powerful Accumulators to Calculate, Summarize & Analyze Your Data

WHAT ARE ACCUMULATOR OPERATORS?

Accumulator operators are used in the $group stage (and other stages) to perform calculations on the grouped documents and return a single result.

KEY CONCEPTS

- Work inside $group stage.
- Operate on a set of documents.
- Return a single value per group.
- Useful for summaries, analytics & reports.

QUICK FACTS

- All accumulators are used in $group.
- Can also be used in $setWindowFields.
- Ignore null values (except $first and $last).
- $count counts the number of documents.

PRO TIP

Use accumulators to turn raw data into meaningful insights with less code and more power!

MOST COMMON ACCUMULATOR OPERATORS

$sum

Calculates the sum of a numeric field.

Example

```
{ $sum: "$amount" }
```

Sample Result: 1850

$avg

Calculates the average of a numeric field.

Example

```
{ $avg: "$amount" }
```

Sample Result: 370

$min

Returns the minimum value of a field.

Example

```
{ $min: "$amount" }
```

Sample Result: 50

$max

Returns the maximum value of a field.

Example

```
{ $max: "$amount" }
```

Sample Result: 1200

$first

Returns the first value of a field in the group.

Example

```
{ $first: "$name" }
```

Sample Result: "Alice"

$last

Returns the last value of a field in the group.

Example

```
{ $last: "$name" }
```

Sample Result: "Charlie"

$count

Counts the number of documents.

Example

```
{ $count: {} }
```

Sample Result: 15

SAMPLE DATA (Collection: sales)

_id	category	item	amount	date
1	Electronics	Laptop	1200	2024-06-01
2	Electronics	Mobile	800	2024-06-02
3	Electronics	Headphones	150	2024-06-03
4	Clothing	T-Shirt	20	2024-06-01
5	Clothing	Jeans	40	2024-06-02
6	Electronics	Watch	1200	2024-06-03

GROUP BY CATEGORY USING ACCUMULATORS

```
db.sales.aggregate([
  { $group: {
      _id: "$category",
      totalAmount: { $sum: "$amount" },
      avgAmount: { $avg: "$amount" },
      minAmount: { $min: "$amount" },
      maxAmount: { $max: "$amount" },
      firstItem: { $first: "$item" },
      lastItem: { $last: "$item" },
      totalCount: { $count: {} }
  }}
])
```

AGGREGATION RESULT

_id (category)	totalAmount ($sum)	avgAmount ($avg)	minAmount ($min)	maxAmount ($max)	firstItem ($first)	lastItem ($last)	totalCount ($count)
Electronics	3350	837.5	150	1200	Laptop	Watch	4
Clothing	60	30	20	40	T-Shirt	Jeans	2

$first and $last depend on the order of documents. Use $sort before $group for meaningful results.

MORE EXAMPLES

Total Sales Amount

```
{ $group: {
    _id: null,
    totalSales: { $sum: "$amount"}
}}
```

Result: 3410

Average Order Amount

```
{ $group: {
    _id: null,
    avgOrder: { $avg: "$amount" }
}}
```

Result: 568.33

Minimum Order Amount

```
{ $group: {
    _id: null,
    minOrder: { $min: "$amount" }
}}
```

Result: 20

Maximum Order Amount

```
{ $group: {
    _id: null,
    maxOrder: { $max: "$amount" }
}}
```

Result: 1200

Total Orders Count

```
{ $group: {
    _id: null,
    totalOrders: { $count: {} }
}}
```

Result: 6

BEST PRACTICES

- Always use $match early to filter data.
- Use $sort before $group when using $first or $last.
- Choose the right accumulator for accurate results.
- Combine accumulators for powerful analytics.
- Test with small datasets before scaling.

Accumulators turn groups into insights – Use them to summarize, analyze & win with your data!

MongoDB $push & $addToSet

Collect Values into Arrays During Aggregation

WHAT ARE $PUSH & $ADDTOSET?

$push and $addToSet are accumulator operators used in the $group stage to collect values into arrays.

KEY CONCEPTS

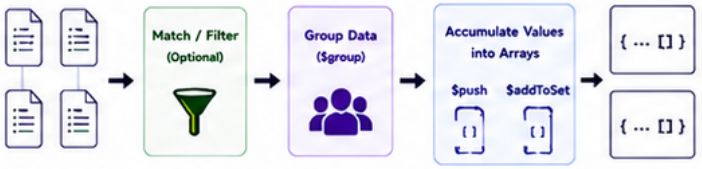
$push
Adds all values to an array. Duplicates are allowed.

$addToSet
Adds only unique values to an array. Duplicates are removed automatically.

QUICK FACTS

- Work only in $group stage.
- $push keeps duplicates.
- $addToSet keeps unique values.
- Both create arrays from values.
- Can be used with any field type.

HOW THEY WORK (IN THE $GROUP STAGE)



$push collects all values. $addToSet collects only unique values.

EXAMPLE: COLLECT VALUES INTO ARRAYS

Sample Data (Collection: orders)

_id	customer	item	amount
1	Alice	Pen	10
2	Alice	Notebook	50
3	Alice	Pen	10
4	Bob	Pen	5
5	Bob	Pencil	5
6	Bob	Pen	5

Group by customer

Using $push (Allows Duplicates)

```
db.orders.aggregate([
  { $group: {
      _id: "$customer",
      items: { $push: "$item" },
      amounts: { $push: "$amount" }
  }
])
```

Result

_id	items	amounts
Alice	["Pen", "Notebook", "Pen"]	[10, 50, 10]
Bob	["Pen", "Pencil", "Pen"]	[5, 5, 5]

Using $addToSet (Unique Values Only)

```
db.orders.aggregate([
  { $group: {
      _id: "$customer",
      items: { $addToSet: "$item" },
      amounts: { $addToSet: "$amount" }
  }
])
```

Result

_id	items	amounts
Alice	["Pen", "Notebook"]	[10, 50]
Bob	["Pen", "Pencil"]	[5]

$PUSH SYNTAX

```
db.collection.aggregate([
  {
    $group: {
      _id: <group_by_field>,
      <arrayField>: { $push: "<field_to_add>"}
    }
  }
])
```

Adds every value to the array (including duplicates).

$ADDTOSET SYNTAX

```
db.collection.aggregate([
  {
    $group: {
      _id: <group_by_field>,
      <arrayField>: { $addToSet: "<field_to_add>"}
    }
  }
])
```

Adds only unique values to the array.

MORE EXAMPLES

Collect Items by Category

```
db.orders.aggregate([
  { $group: {
      _id: "$category",
      allItems: { $push: "$item" },
      uniqueItems: { $addToSet: "$item" }
  }
])
```

Result (Example)

_id	allItems	uniqueItems
Stationery	["Pen","Pencil","Pen"]	["Pen","Pencil"]
Electronics	["Mobile","Laptop","Mobile"]	["Mobile","Laptop"]

Collect Amounts by Customer

```
db.orders.aggregate([
  { $group: {
      _id: "$customer",
      allAmounts: { $push: "$amount" },
      uniqueAmounts: { $addToSet: "$amount" }
  }
])
```

Result (Example)

_id	allAmounts	uniqueAmounts
Alice	[10, 50, 10]	[10, 50]
Bob	[5, 5, 5]	[5]

WHEN TO USE?

Use $push When:
- You need all values.
- Duplicates are important.
- Order of values may matter.

Use $addToSet When:
- You need only unique values.
- Duplicates are not needed.
- Perfect for distinct lists.

PRO TIP

Use $push when you need all values (including duplicates). Use $addToSet when you need only unique values.

NESTED EXAMPLE

```
db.orders.aggregate([
  { $group: {
      _id: "$customer",
      details: { item: "$item", amount: "$amount" } },
      uniqueItems: { $addToSet: "$item" }
  }
])
```

Result

_id	details	uniqueItems
Alice	{ item:"Pen",amount:10}, {item:"Notebook",amount:50}, {item:"Pen",amount:10}]	["Pen","Notebook"]
Bob	{ item:"Pen",amount:5}, {item:"Pencil",amount:5}, {item:"Pen",amount:5}]	["Pen","Pencil"]

EDGE CASES

- $addToSet with null adds null once.
- $push preserves the order of input documents.
- $addToSet order is not guaranteed.
- Works with strings, numbers, documents, arrays, etc.
- Use $sort before $group if order is important.

BEST PRACTICES

- Use $match early to filter data.
- Use $sort before $group for meaningful order (with $push).
- Choose $push for complete list, $addToSet for distinct list.
- Test with small datasets first.

$push and $addToSet help you collect values into arrays—one with duplicates, one with unique values!

MongoDB Aggregation Pipeline

Transform, Analyze & Get Powerful Insights from Your Data

WHAT IS AGGREGATION PIPELINE?

Aggregation Pipeline is a framework for data processing in MongoDB. It allows you to filter, group, transform & analyze data through a series of stages.

KEY BENEFITS

- Powerful data filtering and transformation
- Group and summarize data easily
- Join data from multiple collections
- Create reports and advanced analytics
- Better performance and flexibility

PIPELINE RULES

- Pipeline starts with [and ends with].
- Stages are executed in the order you define.
- Each stage takes input from the previous stage.
- Output of last stage is the final result.

PRO TIP

Start with $match to reduce data early. This improves performance of the pipeline.

AGGREGATION PIPELINE FLOW

1. **$match** — Filter documents that match conditions
2. **$group** — Group documents and perform aggregations
3. **$project** — Reshape documents, include/exclude or rename fields
4. **$sort** — Sort documents by one or more fields
5. **$limit** — Limit the number of documents passed
6. **$lookup** — Join with another collection (left outer join)
7. **$unwind** — Deconstruct an array field into multiple documents
8. **RESULT** — Final output of the pipeline

These stages are commonly used. You can use as many stages as needed in your pipeline.

1. $match (Filter)

Filters documents based on condition.

```
{ $match:
  { status: "A",
    amount: { $gt: 100 } }
} )
```

Example: Get documents with amount > 100.

2. $group (Group & Aggregate)

Groups documents by a field and calculates values.

```
{ $group: {
   _id: "$category",
   totalAmount: { $sum: "$amount" },
   count: { $sum: 1 }
} )
```

Example: Total amount & count per category.

3. $project (Reshape)

Include, exclude or rename fields in output.

```
{ $project: {
   _id: 0,
   category: "$_id",
   total: "$totalAmount",
   count: 1
  }
```

Example: Show only required fields with new names.

4. $sort (Sort)

Sorts documents in ascending (1) or descending (-1) order.

```
{ $sort: { total: -1 } }
```

Example: Sort by total in descending order.

5. $limit (Limit)

Limits the number of documents passed.

```
{ $limit: 5 }
```

Example: Return only top 5 documents.

6. $lookup (Join)

Join documents from another collection.

```
{ $lookup: {
    from: "users",
    localField: "userId",
    foreignField: "_id",
    as: "userInfo"
} }
```

Example: Join user details from users collection.

7. $unwind (Deconstruct Array)

Deconstructs an array field into multiple documents.

```
{ $unmind: "$items" }
```

Example: Create a document for each item in the array.

EXAMPLE: SALES ANALYTICS PIPELINE

```
[
  { $match: { status: "A" } },
  { $proup: { _id: "$category",
      totalAmount: { $sum: "$amount" },
      count: { $sum: 1 } } },
  { $project: { _id: 0, category: "$_id",
      total: "$totalAmount", count: 1 } },
  { $sort: { total: -1 } },
  { $limit: 5 }
]
```

STEP BY STEP FLOW

1. $match → Filter data (status = A)
2. $group → Group by category, calculate totalAmount & count
3. $project → Reshape output fields
4. $sort → Sort by total (High → Low)
5. $limit → Show top 5 results

SAMPLE DATA (Collection: sales)

_id	category	item	amount	status
1	Electronics	Laptop	1200	A
2	Electronics	Mobile	800	A
3	Clothing	T-Shirt	20	A
4	Electronics	Headphones	150	B
5	Clothing	Jeans	40	A
6	Electronics	Watch	1200	A

PIPELINE RESULT (Top 5 Categories)

category	total	count
Electronics	3200	3
Clothing	60	2

GOOD PRACTICES

- Always use $match early.
- Project only needed fields.
- Use indexes for better performance.
- Test each stage separately.
- Keep pipeline simple and readable.

Aggregation Pipeline turns raw data into meaningful insights. Combine stages in order to get the results you need!

MongoDB Indexes

Faster Queries, Better Performance

WHAT IS AN INDEX?

An index improves the speed of read operations on a collection. It works like an index in a book – helps find data quickly without scanning all documents.

KEY BENEFITS

- **Faster Queries** — Speeds up find() operations
- **Better Performance** — Reduces response time
- **Efficient Sorting** — Improves sort() performance
- **Optimized Lookups** — Speeds up joins and filters

QUICK FACTS

- Indexes improve read speed, not write speed.
- Too many indexes can slow down inserts/updates.
- Use indexes on fields that are frequently queried or sorted.
- _id field is indexed by default.

PRO TIP

Create indexes on fields used in $match, $sort, $group, $lookup and frequently in find() queries.

CREATE INDEX & USE IN QUERIES

1. Create Index

```
db.users.createIndex({
  name: 1
})
```

Options:

```
1     : Ascending Order
-1    : Descending Order
text  : Text Index
unique: true : Unique Index
```

2. Query with Index

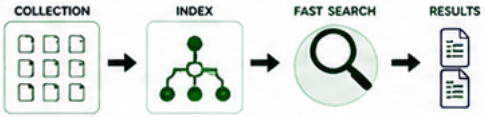
```
db.users.find({ name: "Aman" })
```

3. View Indexes

```
db.users.getIndexes()
```

4. Drop Index

```
db.users.dropIndex("name_1")
```



MongoDB uses indexes to find data faster without scanning the whole collection.

HOW INDEX IMPROVES QUERY SPEED

Without Index (COLLSCAN)

```
db.users.find({ name: "Aman" })
```

MongoDB scans every document in the collection.

Slow

With Index (IXSCAN)

```
db.users.find({ name: "Aman" })
```

MongoDB uses the index to jump directly to matching documents.

Fast

Index Scan (IXSCAN) is much faster than Collection Scan (COLLSCAN).

TYPES OF INDEXES

1. Single Field Index

Index on one field.

```
db.users.createIndex({
  email: 1
})
```

Use: Equality & range queries on a single field.

2. Compound Index

Index on multiple fields.

```
db.users.createIndex({
  name: 1,
  age: -1
})
```

Use: Queries on multiple fields together.

3. Unique Index

Ensures all values are unique.

```
db.users.createIndex({
  email: 1
}, { unique: true })
```

Use: Queries on multiple values.

4. Text Index

For text search queries.

```
db.articles.createIndex({
  title: "text",
  content: "text"
})
```

Use: Prevent duplicate $text operator.

5. Hashed Index

Indexes hashed value.

```
db.users.createIndex({
  userId: "hashed"
})
```

Use: Text search using high cardinality fields.

6. TTL Index

Automatically removes documents after time.

```
db.sessions.createIndex({
  expiresAt: 1
}, { expireAfterSeconds: 3600 })
```

Use: Session management, logs, temporary data.

CHECK INDEX USAGE (EXPLAIN)

```
db.users.find({ name: "Aman" }).explain("executionStats")
```

```
"executionStats" : {
  "executionStages" : {
    "stage" : "IXSCAN",
    "indexName" : "name_1",
    "nReturned" : 1,
    "totalKeysExamined" : 1,
    "totalDocsExamined" : 1
  }
}
```

Stage	Meaning
IXSCAN	Used Index
COLLSCAN	Full Collection Scan
FETCH	Fetch Documents

Use explain() to check if your query is using an index.

EXAMPLE

Collection: users

_id	name	age	city
1	Aman	25	Delhi
2	Priya	30	Mumbai
3	Rahul	28	Pune
4	Neha	22	Delhi
5	John	27	Bangalore

Create Index on age:

```
db.users.createIndex({ age: 1 })
```

Query:

```
db.users.find({ age: { $gt: 25 } })
```

Result: Uses index for fast search

PERFORMANCE IMPACT

Before Index (COLLSCAN)

Documents Examined: 1,000,000

Keys Examined: 0

Execution Time: 1250 ms

Slow

After Index (IXSCAN)

Documents Examined: 12

Keys Examined: 12

Execution Time: 8 ms

Fast

USE INDEXES ON

- Fields used in $match
- Fields used in $sort
- Fields used in $group
- Fields used in $lookup (localField)
- Frequently searched fields

BEST PRACTICES

- Create indexes on frequently queried fields.
- Avoid indexing fields with low cardinality.
- Use compound indexes carefully.
- Monitor index size and remove unused indexes.
- Test with explain() to verify index usage.

Indexes make your MongoDB queries faster and your application faster! Use them smartly for best performance.

MongoDB Compound Index

Index on Multiple Fields for Faster & Smarter Queries

WHAT IS COMPOUND INDEX?

A compound index indexes multiple fields in a specific order. It improves query performance when queries use those fields together.

KEY BENEFITS

- Faster queries on multiple fields
- Improves sorting performance
- Efficient for filtering + sorting
- Reduces in-memory sorting
- Better overall performance and scalability

QUICK FACTS

- Order of fields matters.
- Supports equality, range & sort operations.
- Helpful for complex queries.
- Uses less memory than separate indexes.

PRO TIP

Place fields with equality match first, then range fields, and then sort fields for best performance.

CREATE COMPOUND INDEX

Syntax

```
db.collection.createIndex({
  field1: 1,
  field2: -1,
  field3: 1
})
```

Example

```
db.products.createIndex({
  category: 1,     // Field 1 (Ascending)
  price: -1,       // Field 2 (Desending)
  name: 1          // Field 3 (Ascending)
})
```

How It Works

- MongoDB sorts documents first by Field 1.
- Then by Field 2 if Field 1 values are same.
- Then by Field 3 if both Field 1 and Field 2 are same.
- This order is used to quickly locate matching documents.

Index Structure (Left to Right)

Field 1 (category)	→	Field 2 (price)	→	Field 3 (name)
Equality Match		Range / Sort		Sorting / Tie-breaker

Compound index follows the leftmost prefix rule.

FIELD ORDER MATTERS

Query Condition	Uses Index?	Why?
`{ category: "Electronics", price:500 }`	Yes	Uses (category, price)
`{ category: "Electronics" }`	Yes	Uses (category) part
`{ category: "Electronics", price: { $gt: 100 } }`	Yes	Uses both fields
`{ price: 500 }`	No	Starts from second field
`{ price: 500, category: "Electronics" }`	No	Order matters
`{ name: "Laptop" }`	No	Skips previous fields

SORTING WITH COMPOUND INDEX

With Index (category: 1, price: -1)

```
db.products.find({ category: "Electronics" })
  .sort({ price: -1 })
```

Index Used

Sorted by price in descending order without extra sorting.

Without Index (or wrong order)

```
db.products.find({ category: "Electronics" })
  .sort({ price: -1 })
```

Index Not Used

MongoDB must sort in memory (extra work).

EXAMPLE DATA

Collection: products

_id	category	price	name
1	Electronics	1200	Laptop
2	Electronics	800	Mobile
3	Electronics	1200	Camera
4	Clothing	500	T-Shirt
5	Clothing	300	Jeans
6	Electronics	500	Earbuds
7	Electronics	800	Tablet

Create Compound Index

```
db.products.createIndex({
  category: 1,
  price: -1
})
```

Index Key Pattern

```
{ category: 1, price: -1 }
```

category ↑ (Ascending) → price ↓ (Descending)

QUERY EXAMPLES

		Index Used?
Query	`db.products.find({ category: "Electronics", price: { $gte: 500 } }).sort({ price: -1 })`	Yes (Best Use)
Query	`db.products.find({ category: "Electronics" })`	Yes (Partial Use)
Query	`db.products.find({ price: { $gte: 500 } })`	No (Wrong Order)

CHECK WITH explain()

```
db.products.find({ category: "Electronics", price: { $gte: 500 } })
           .sort({ price: -1 })
    .explain("executionStats")
```

```
"executionStats": {
  "executionStages": {
    "stage": "IXSCAN",
    "indexName": "category_1_price_-1",
    "totalKeysExamined": 5,
  }  "nReturned": 3
})
```

Using Index (IXSCAN)

COMMON USE CASES

- Filter by category and sort by price (e.g., e-commerce)
- Find users by status and lastLogin date
- Search products by brand and rating
- Reports with group + sort operations

BEST PRACTICES

- Put equality fields first, then range fields, then sort fields.
- Do not create too many indexes.
- Avoid redundant indexes.
- Monitor index usage with explain( ) and $indexStats.
- Test performance before and after index creation.

LEFTMOST PREFIX RULE

Index `{ A: 1, B: 1, C: 1 }` can be used for:

- ✓ `{ A: value }`
- ✓ `{ A: value, B: value }`
- ✓ `{ A: value, B: { $gt: 10 } }`
- ✓ `{ A: value, B: value, C: value }`
- ✗ `{ B: value }` (Not used)

PERFORMANCE IMPACT

Without Compound Index	With Compound Index
Documents Scanned: 100000	Documents Scanned: 120
Execution Time: 1250 ms	Execution Time: 8 ms
Slow	Fast

REMEMBER

The right compound index can reduce query time from seconds to milliseconds!

USEFUL COMMANDS

```
db.collection.createIndex({ field1: 1, field2: -1 })   // Create Index
db.collection.getIndexes( )                            // List Indexes
db.collection.dropIndex( "field1_1_field2_-1" )        // Drop Index
db.collection.stats( )                                 // Index Size & Usage
```

Compound Index = Smart Indexing + Right Field Order = Super Fast Queries!

MongoDB Unique Index

Ensure Unique Values & Prevent Duplicates Automatically

WHAT IS UNIQUE INDEX?

A Unique Index ensures that all values in the indexed field are unique across the collection. It prevents duplicate data.

KEY BENEFITS

- Prevents duplicate values in a field
- Maintains data integrity & accuracy
- Faster queries on unique fields
- Useful for fields like Email, Username, Phone, etc.

CREATE UNIQUE INDEX

Syntax

```
db.collection.createIndex(
  { fieldName: 1 },
  { unique: true }
)
```

Example: Unique Index on Email

```
db.users.createIndex(
  { email: 1 },
  { unique: true }
)
```

→ Unique Index Created on "email" field

How Unique Index Works

Insert Document `{ email: "test@example.com" }` → Unique Index on email → Insert Another Document `{ email: "test@example.com" }` → Duplicate Value Error Not Allowed!

If a duplicate value is inserted, MongoDB throws an error and the document is not inserted.

EXAMPLE: UNIQUE INDEX IN ACTION

1. Insert Unique Values (Allowed)

_id	email	username	
1	alice@example.com	alice123	✓
2	bob@example.com	bob456	✓
3	charlie@example.com	charlie789	✓

```
db.users.insertMany([
  { email: "alice@example.com", username: "alice123" },
  { email: "bob@example.com", username: "bob456" },
  { email: "charlie@example.com", username: "charlie789" }
])
```

2. Insert Duplicate Value (Blocked)

_id	email	username	
–	alice@example.com	alice001	✗

```
E11000 duplicate key error
  collection: users index: email_1
  dup key: { email: "alice@example.com" }
```

Duplicate email is not allowed! MongoDB prevents it automatically.

QUICK FACTS

- Only unique values are allowed.
- One null value is allowed (by default).
- Case-sensitive by default.
- Compound unique indexes are also supported.

COMMON UNIQUE INDEX USE CASES

- **Email** — Ensure each user has a unique email address.
- **Username** — Ensure each user has a unique username.
- **Phone Number** — Ensure each phone number is unique in the system.
- **Employee ID** — Ensure each employee has a unique ID.
- **Order Number** — Ensure each order number is unique in orders.

COMPOUND UNIQUE INDEX

Create unique index on multiple fields.

```
db.users.createIndex(
  { email: 1, username: 1 },
  { unique: true }
)
```

email + username = Combination must be unique.

INDEX INFO

View all indexes on a collection.

```
db.users.getIndexes()
```

Drop an index.

```
db.users.dropIndex("email_1")
```

_id index is unique by default.

CONSTRAINT (RULE)

For a unique index:

- No duplicate values allowed
- Only one null value allowed
- Enforced at the index level
- Ensures data consistency

PRO TIP

Use Unique Index on fields like email, username, or mobile number to keep your data clean and reliable.

EXAMPLE: REAL WORLD SCENARIO

In a signup system, we create a unique index on email to prevent users from registering with the same email.

User Signup (email: user@example.com) → Check Unique on email → If Unique Allow Signup / If Duplicate Reject Signup

PERFORMANCE IMPACT

Unique indexes improve performance for queries on indexed fields.

- **Without Index (COLLSCAN)** — Slow query performance, Full collection scan → Slow
- **With Unique Index (IXSCAN)** — Fast query performance, Uses index for lookup → Fast

EXAMPLE QUERIES (uses unique index)

```
// Find user by email (fast lookup)
db.users.find({ email: "alice@example.com" })

// Find user by username (if indexed uniquely)
db.users.find({ username: "alice123" })
```

Queries on unique indexed fields are super fast!

BEST PRACTICES

- Create unique index on fields that must be unique.
- Use unique index for email, username, phone, etc.
- Test for duplicates before creating unique index.
- Use compound unique index when needed.
- Monitor index usage and performance.

Unique Index = Data Integrity + Duplicate Prevention + Faster Queries!

MongoDB Text Index

Powerful Full-Text Search for Your Data

WHAT IS TEXT INDEX?

Text Index enables you to perform full-text search on string content in a collection. It searches words, not just exact matches.

KEY BENEFITS

- Search text in large content
- Find documents by keywords
- Rank results by relevance ($textScore)
- Supports language stemming & stop words
- Great for blogs, articles, comments, products, etc.

QUICK FACTS

- Only one text index per collection.
- Text index can include multiple string fields.
- Search is case-insensitive.
- Ignores common stop words (e.g., the, is, an).
- Returns results ranked by relevance.

PRO TIP

Use text index for searching articles, blog posts, product descriptions, comments, manuals, FAQs, etc.

CREATE TEXT INDEX

Syntax

```
db.collection.createIndex({
  field1: "text",
  field2: "text",
  field3: "text"
})
```

Example: Text Index on title and content

```
db.articles.createIndex({
  title: "text",
  content: "text",
  tags: "text"
})
```

→ Text Index Created on multiple fields

How Text Search Works

User Search	→	Text Index	→	Matching & Scoring	→	Search Results
Search Words / Keywords		Indexed Words from Documents		Calculate Relevance ($textScore)		Ranked by Relevance

MongoDB breaks text into words, removes stop words, applies stemming, and ranks results by relevance.

TEXT SEARCH QUERY

Basic Text Search with $text

```
db.articles.find(
  { $text: { $search: "mongodb database" } }
)
```

Project Relevance Score ($textScore)

```
db.articles.find(
  { $text: { $search: "mongodb database" } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
```

Search with Language

```
db.articles.find(
  { $text: { $search: "mongodb database",
             $language: "english" } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
```

Important:

- Only one text index per collection.
- Use $meta: "textScore" to sort by relevance.
- Search is not case-sensitive.

EXAMPLE DATA (articles Collection)

_id	title	content	tags
1	MongoDB Guide	Learn MongoDB database in easy steps	["mongodb","database","guide"]
2	Full Text Search in MongoDB	Text search using $text and text index	["mongodb","search","text"]
3	Node.js with MongoDB	Build real apps with Node.js and MongoDB	["nodejs","mongodb","backend"]
4	Database Indexing	Types of indexes in MongoDB	["database","index","performance"]
5	MongoDB Aggregation	Aggregation framework in MongoDB	["mongodb","aggregation","pipeline"]

SAMPLE SEARCH & RESULTS

Search: "mongodb database"

_id	title	score ($textScore)
1	MongoDB Guide	2.35 ★★★★★
4	Database Indexing	1.80 ★★★★☆
2	Full Text Search in MongoDB	1.45 ★★★★☆
3	Node.js with MongoDB	0.90 ★★★☆☆
5	MongoDB Aggregation	0.60 ★☆☆☆☆

Results are ranked by relevance ($textScore). Higher score = More relevant.

SEARCH EXAMPLE: DIFFERENT QUERIES

Query	Description	Example
Single Word	Search single keyword	{ $text: { $search: "mongodb" } }
Multiple Words	Search multiple keywords	{ $text: { $search: "mongodb index" } }
Phrase Search	Search exact phrase	{ $text: { $search: "text search" } }
Exclusion	Exclude a word	{ $text: { $search: "mongodb -nodejs" } }
Language	Search with specific language	{ $text: { $search: "mongodb", $language: "english" } }

SUPPORTED FEATURES

Stemming	Stop Words	Scoring	Multiple Fields	Languages
Finds variations of a word (run, running, runs)	Ignores common words (the, is, and, or, etc.)	Ranks results by relevance using $textScore	Create text index on multiple string fields	Supports multiple languages for search

INDEX INFO

```
db.articles.getIndexes()
[
  "v" : 2,
  "key" : { "_id" : 1 },
  "name" : "_id_",
  "v" : 2,
  "key" : { "title" : "text", "content" : "text", "tags" : "text" },
  "name" : "title_content_tags_text",
  "weights" : { "title" : 1, "content" : 1, "tags" : 1 }
]
```

LIMITATIONS

- Only one text index per collection.
- Works only on string fields.
- Not suitable for prefix search (use regex or autocomplete).
- Does not support partial matching within a word.

USE CASES

- Blog / Article Search
- Product Search
- Documentation Search
- FAQ / Help Center Search
- Comments & Reviews Search

BEST PRACTICES

- Create text index on important fields only.
- Use $textScore to sort by relevance.
- Test with different keywords.
- Combine with other filters for better results.
- Keep content clean and meaningful.

Text Index + $text + $textScore = Smarter Search, Better Results, Happy Users!

MongoDB Explain Query

Understand How Your Query Runs & Improve Performance

WHAT IS EXPLAIN()?

explain() shows how MongoDB executes a query. It helps you analyze the query plan and performance.

KEY BENEFITS

- Understand query execution
- Find performance bottlenecks
- Compare index vs collection scan
- Reduce response time
- Optimize database performance

QUICK FACTS

- explain() does NOT execute the query.
- It only returns the execution plan.
- Works for find(), aggregate(), update(), delete(), etc.
- Use it before & after creating index to see the improvement.

PRO TIP

Always check 'totalDocsExamined' and 'executionTimeMillis'. Lower is Better! 🚀

HOW TO USE

Syntax

```
db.collection.find(<query>).explain(<mode>)
```

Example

```
db.users.find({ age: { $gt: 18 } }).explain("executionStats")
```

Explain Modes

Mode	Description
queryPlanner	Returns the query plan.
executionStats	Returns execution statistics (most useful).
allPlansExecution	Returns all execution plans with stats.

"allPlansExecution" is helpful to see why a plan was chosen.

COMMON EXECUTION STAGES

COLLSCAN (Collection Scan)

MongoDB scans every document in the collection.

IXSCAN (Index Scan)

MongoDB uses index to find the matching documents.

IXSCAN is faster than COLLSCAN!

SAMPLE OUTPUT (executionStats)

```
{
  "queryPlanner" : {
    "winningPlan" : {
      "stage" : "IXSCAN",
      "keyPattern" : { "age" :1 },
      "indexName" : "age_1",
      "direction" : "forward"
    }
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 25,
    "executionTimeMillis" : 3,
    "totalKeysExamined" : 25,
    "totalDocsExamined" : 25,
    "executionStages" : {
      "stage" : "IXSCAN",
      "nReturned" : 25,
      "works" : 26,
      "advanced" : 25,
      "needTime" : 0,
      "needYield" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1
    }
  }
}
```

Field	Meaning
nReturned	Documents returned by the query.
executionTimeMillis	Time taken to execute the query (ms).
totalKeysExamined	Total index keys scanned.
totalDocsExamined	Total documents scanned.
stage	Type of stage used (COLLSCAN / IXSCAN).
IndexName	Index used in the query.

Good Query → Low executionTimeMillis, Low totalDocsExamined, and uses IXSCAN.

COMPARE: COLLSCAN vs IXSCAN

Metric	COLLSCAN	IXSCAN
Stage	COLLSCAN	IXSCAN
Indexes Used	No	Yes
Keys Examined	0	> 0
Docs Examined	High (All Docs)	Low (Matched Docs)
Execution Time	Slow	Fast
Performance	Poor	Excellent

Example: COLLSCAN

```
db.users.find({ age: 25 })
  .explain("executionStats")

stage: "COLLSCAN"
totalDocsExamined: 10000
executionTimeMillis: 120
```

Example: IXSCAN

```
db.users.find({ age: 25 })
  .explain("executionStats")

stage: "IXSCAN"
totalDocsExamined: 10
executionTimeMillis: 2
```

WHAT TO CHECK?

- Check winningPlan.stage
- Prefer IXSCAN over COLLSCAN
- Check totalDocsExamined
- Check executionTimeMillis
- Compare before & after index

NEED INDEX?

If you see COLLSCAN and high Docs Examined, create an index.

```
db.users.createIndex({ age: 1 })
```

Re-run explain() and you should see IXSCAN and better stats!

TYPICAL WORKFLOW

1. Write Your Query
2. Run explain()
3. Analyze Output
   - Check Stage
   - Check Stats
4. Optimize
   - Add Index
   - Rewrite Query
5. Re-check
   Run explain() again and verify improvement

BEST PRACTICES

- Always test queries with explain() during development.
- Create indexes for fields used in filter, sort, and join.
- Avoid COLLSCAN on large collections.
- Monitor queries in production using explain() & logs.
- Optimize for low execution time and low docs examined.

USEFUL COMMANDS

```
db.collection.find().explain("queryPlanner")              // Show Query Plan
db.collection.find().explain("executionStats")            // Show Execution Stats
db.collection.find().explain("allPlansExecution")         // Show All Plans
db.collection.aggregate(pipeline).explain("executionStats") // Aggregate Explain
db.collection.getIndexes()                                // List Indexes
```

explain() = Visibility + Insights = Faster Queries + Better Performance!

MongoDB Schema Design

Design Smarter Schemas for Scalable & High Performance Applications

WHAT IS SCHEMA DESIGN?

Schema design is how you structure your data in MongoDB. A good schema improves performance, scalability and flexibility.

KEY PRINCIPLES

- Model Data Around Queries
- Denormalize for Performance
- Avoid Joins (Use Embedding)
- Reference When Needed
- Keep it Simple & Flexible

QUICK FACTS

- MongoDB is schema-less, not schema-free.
- You can change your schema anytime.
- Embedding reduces queries.
- Referencing avoids data duplication.
- Choose based on access patterns.

PRO TIP

Always model your data based on how you read it most, not how you write it.

EMBEDDING vs REFERENCING

EMBEDDING (Denormalization)

Store related data inside a single document.

```
{
 _id: ObjectId("...")
 name: "Order 101"
 orderDate: ISODate("...")
 items: [
   { product: "Laptop", qty: 1,
     price: 60000 },
   { product: "Mouse", qty: 2,
 } ]
 total: 61000
}
```

Use When:

- Data is closely related
- One-to-Few relationships
- You read the data together
- Data size is not too large

Faster Reads, Fewer Queries

REFERENCING (Normalization)

Store related data in separate collections.

```
orders Collection
{
 _id: ObjectId("...")
 name: "Order 101"
 orderDate: ISODate("...")
 items: [
   ObjectId("p1"),
   ObjectId("p2")
 }
 total: 61000
}
```

```
products Collection
{
 _id: ObjectId("p1")
 name: "Laptop"
 price: 60000
}
{
 _id: ObjectId("p2")
 name: "Mouse"
 price: 500
}
```

Use When:

- Data is reused in many places
- Many-to-Many relationships
- Data size is very large
- You write data in one place, read in another

Less Duplication, More Flexible

RELATIONSHIP PATTERNS

ONE-TO-ONE

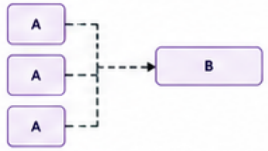
A references B and B references A (rare in MongoDB)

ONE-TO-MANY

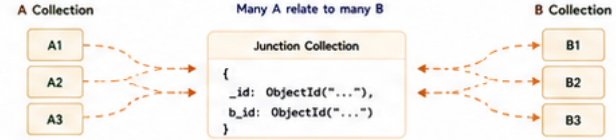
One A has many B

MANY-TO-ONE

Many A refer to one B



MANY-TO-MANY



DATA MODEL EXAMPLES

EMBEDDED (One-to-Few)

Order → Items

```
{
 _id: ObjectId("order1"),
 customer: "John",
 items: [
   { product: "Laptop", qty: 1 },
   { product: "Mouse", qty: 2 }
 ],
 total: 61000
}
```

Good for orders with few items.

REFERENCED (Many-to-One)

Orders → Customers

```
orders Collection
{
 _id: ObjectId("order1"),
 customer_id: ObjectId("c1"),
 total: 61000
}
```

```
customers Collection
{
 _id: ObjectId("c1"),
 name: "John",
 email: "john@example.com"
}
```

Good when customer data is shared across many orders.

MANY-TO-MANY (with Junction)

Students ↔ Courses

```
students
{
 _id: ObjectId("s1"),
 name: "Alice"
}
```

```
courses
{
 _id: ObjectId("c1"),
 title: "MongoDB 101"
}
```

```
enrollments (Junction)
{
 student_id: ObjectId("s1"),
 course_id: ObjectId("c1"),
 enrolled_on: ISODate("...")
}
```

ONE-TO-ONE (Example)

User ↔ Profile

```
users
{
 _id: ObjectId("u1"),
 name: "Alice",
 profile_id: ObjectId("p1")
}
```

```
profiles
{
 _id: ObjectId("p1"),
 address: "India",
 phone: "9876543210"
}
```

WHEN TO USE WHAT?

Use Case	Use	Why?
Small, related data	Embedding	Faster reads
Large, shared data	Referencing	Avoid duplication
Many-to-many	Referencing + Junction	Flexible & scalable
Frequent updates on child data	Referencing	Avoid large embedded docs

DATA MODEL CHECKLIST

- [ ] What are my most frequent queries?
- [ ] Do I need the related data together?
- [ ] How large can this data grow?
- [ ] Will I frequently update the child data?
- [ ] Is the data reused in multiple places?

BEST PRACTICES

- Design for your queries, not for your data.
- Prefer embedding for read-heavy workloads.
- Use referencing for write-heavy or shared data.
- Avoid deep nesting and huge documents (> 16MB limit).
- Keep your schema simple, flexible and scalable.

USEFUL COMMANDS

```
db.collection.find()                          // Find documents
db.collection.aggregate([...])                // Aggregation pipeline
db.collection.createIndex({ field: 1 })       // Create index
db.collection.explain("executionStats")       // See query plan & stats
```

Good Schema = Better Performance + Scalability + Happy Developers!

MongoDB Embedded vs Referenced

Choose the Right Approach for Your Data & Relationships

WHAT IS THE DIFFERENCE?

MongoDB allows you to model relationships in two main ways: Embedding or Referencing. Both have pros & cons!

KEY CONCEPTS

Embedding: Store related data inside a single document (Nested Data).

Referencing: Store related data in separate collections and link using ObjectId.

ObjectId: Unique 12-byte identifier used to reference documents.

WHEN TO USE WHAT?

Embed when:
- Data is tightly related
- Accessed together
- Small & bounded in size

Reference when:
- Data is large or unbounded
- Used independently
- Many-to-many relationships

PRO TIP

Start with Embedding for simplicity and performance. Use Referencing only when necessary!

EMBEDDING

Store Related Data Inside a Single Document (Nested Data)

Example: Blog with Comments (Embedded)

```
{
  _id: ObjectId("64f1..."),
  title: "MongoDB Schema Design",
  author: "CodeSardar",
  comments: [
    {
      user: "User1",
      text: "Great post!",
      createdAt: ISODate("2024-01-01T10:00:00Z")
    },
    {
      user: "User2",
      text: "Very helpful 👍",
      createdAt: ISODate("2024-01-01T11:00:00Z")
    }
  ]
}
```

Blogs Collection

Blog Document

```
_id
title
author
comments: [
  { ... }
  { ... }
]
```

Nested Comments (Inside Blog Document)

Benefits
- Faster reads (one query)
- Better performance
- Atomic updates
- Simpler to query

Limitations
- Document grows in size
- Harder to update individual items
- Not suitable for large/unbounded data
- Can hit 16MB document limit

REFERENCING

Store Related Data in Separate Collections and Link Using ObjectId

blogs Collection

```
{
  _id: ObjectId("64f1..."),
  title: "MongoDB Schema Design",
  author: "CodeSardar",
  commentIds: [
    ObjectId("64f2..."),
    ObjectId("64f3...")
  ]
}
```

comments Collection

```
{
  _id: ObjectId("64f2..."),
  blogId: ObjectId("64f1..."),
  user: "User1",
  text: "Great post!",
  createdAt: ISODate("2024-01-01T10:00:00Z")
}
{
  _id: ObjectId("64f3..."),
  blogId: ObjectId("64f1..."),
  user: "User2",
  text: "Very helpful 👍",
  createdAt: ISODate("2024-01-01T11:00:00Z")
}
```

blogs Collection

Blog Document

```
_id
title
author
commentIds: [
  ObjectId("64f2...")
  ObjectId("64f3...")
]
```

References (ObjectId)

comments Collection

Comment Document

```
_id
blogId (ObjectId)
user
text
createdAt
```

Comment Document

```
_id
blogId (ObjectId)
user
text
createdAt
```

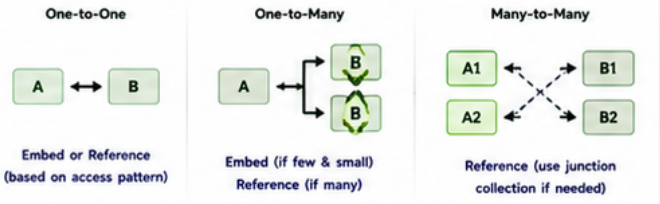
Benefits
- Good for large or growing data
- Supports unbounded data
- Reusability of data
- Better for many-to-many

Limitations
- Slower reads (requires $lookup or multiple queries)
- More complex queries
- Not atomic across collections

COMPARISON AT A GLANCE

Aspect	Embedding	Referencing
Data Location	Inside the same document	In separate collections
Relationships	One-to-Few, One-to-One	One-to-Many, Many-to-Many
Query Performance	Faster (single query)	Slower (joins / multiple queries)
Update	Update whole document	Update individual documents
Best For	Small, tightly related data	Large, loosely related data
Document Size	Limited by 16MB	No size limit
Complexity	Simple	More complex

COMMON RELATIONSHIP PATTERNS



SCHEMA DESIGN CHECKLIST

- Understand your queries
- Identify access patterns
- Estimate data growth
- Choose: Embed or Reference
- Keep it simple
- Measure and optimize

There is no One-Size-Fits-All! Choose the Right Schema Design for Your Application.

MongoDB Validation

Ensure Data Integrity with Schema Validation

WHAT IS VALIDATION?

MongoDB Validation enforces rules on documents before they are inserted or updated in a collection.

KEY BENEFITS

- Maintain consistent schema
- Prevent invalid data
- Improve data quality
- Reduce application errors
- Ensure data integrity

QUICK FACTS

- Validation is applied at collection level.
- Uses $validator with $jsonSchema.
- Works on insert and update operations.
- Existing documents are not validated automatically.
- Use validationLevel to control strictness.

PRO TIP

Always test validation rules thoroughly before applying to production collections.

CREATE COLLECTION WITH VALIDATION

Syntax

```
db.createCollection("collectionName", {
   validator: { $jsonSchema: { /* rules */ } },
   validationLevel: "strict",    // or "moderate"
   validationAction: "error"     // or "warn"
})
```

Example

```
db.createCollection("users", {
   validator: {
      $jsonSchema: { /* see example below */ }
   },
   validationLevel: "strict",
   validationAction: "error"
})
```

validationLevel

- strict : New & updated documents must match the rules.
- moderate : Only new & updated documents are validated. Existing invalid docs allowed.

validationAction

- error : Rejects invalid documents.
- warn : Logs a warning, but allows the operation.

VALIDATION IN ACTION

Valid Document

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "age": 25,
  "status": "active"
}
// Insert: Success
```

Invalid Document ✗ (Missing Required Field)

```
{
  "name": "Bob",
  "age": 30
}

// Error: Fails - email is required
```

Invalid Document ✗ (Wrong Type)

```
{
  "name": "Charlie",
  "email": "charlie@example.com",
  "age": "twenty"
}

// Error: Fails - age must be int
```

$JSONSCHEMA EXAMPLE

Example Schema (users Collection)

```
{
  $jsonSchema: {
    bsonType: "object",
    required: [ "name", "email", "age" ],
    properties: {
      _id: { bsonType: "objectId" },
      name: {
        bsonType: "string",
        description: "Name is required and must be a string"
      },
      email: {
        bsonType: "string",
        pattern: "^.+@.+\..+$",
        description: "Must be a valid email"
      },
      age: {
        bsonType: "int",
        minimum: 0,
        maximum: 120,
        description: "Agee must be between 0 and 120"
      },
      status: {
        bsonType: "string",
        enum: [ "active", "inactive" ],
        description: "Status can be only active or inactive"
      }
    },
    additionalProperties: false
  }
}
```

Key Components

- **bsonType** — Defines the BSON data type.
- **required** — Fields that must be present.
- **properties** — Define rules for each field.
- **minimum / maximum** — Set numeric limits.
- **pattern** — Validate string using regex.
- **enum** — Allow only specific values.
- **additionalProperties** — false = No extra fields allowed.

COMMON VALIDATION RULES

Rule	Description	Example
bsonType	Specify the BSON data type	{ bsonType: "string" }
required	Make field mandatory	{ required: ["name"] }
minimum / maximum	Set number range	{ minimum: 0, maximum: 100 }
minLength / maxLength	Set string length	{ minLength: 3, maxLength: 50 }
pattern	Validate string with regex	{ pattern: "^.+@.+\..+$" }
enum	Allow only specific values	{ enum: ["active", "inactive"] }
additionalProperties	Allow extra fields or not	{ additionalProperties: false }

MODIFY OR REMOVE VALIDATION

Modify Validation

```
db.runCommand({
  collMod: "users",
  validator: { $jsonSchema: { /* new rules */ } },
  validationLevel: "strict",
  validationAction: "error"
})
```

Remove Validation

```
db.runCommand({
  collMod: "users",
  validator: {},
  validationLevel: "off"
})
```

EXAMPLE VALIDATION ERROR

```
{
   "ok" : 0,
   "errmsg" : "Document failed validation",
   "code" : 121,
   "codeName" : "DocumentValidationFailure",
   "errInfo" : {
      "failingDocumentId" : ObjectId("...")
   }
}
```

MongoDB prevents invalid data from entering your collection!

BEST PRACTICES

- Validate at the collection level.
- Use strict mode in production.
- Keep schema simple and clear.
- Test validation rules thoroughly.
- Monitor logs for warnings.

Validation = Clean Data + Fewer Bugs + Better Performance + Happy Users!

MongoDB Atlas

The Official MongoDB Cloud Database Platform

WHAT IS MONGODB ATLAS?

MongoDB Atlas is a fully managed cloud database service built by MongoDB. It handles deployment, scaling, backups, security and maintenance for you.

KEY BENEFITS

- Fully Managed Service
- Auto Scaling
- High Availability
- Global Deployment
- Built-in Security
- Backups & Point-in-Time Recovery
- Monitoring & Alerts
- Pay as You Go

QUICK FACTS

- Hosted on AWS, Azure & GCP
- Multi-Cloud & Multi-Region
- 99.95% Availability SLA
- Supports MongoDB Features
- Secure by Default

PRO TIP

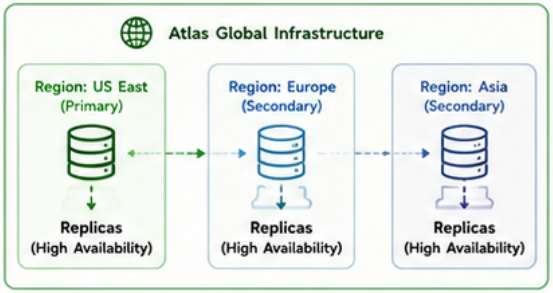
Use Atlas Free Tier to explore MongoDB in the cloud and upgrade as your app grows.

HOW ATLAS WORKS

You	Cluster	Database	Collections	Documents
Create an account on MongoDB Atlas	Deploy a fully managed MongoDB cluster	Create databases inside the cluster	Create collections inside databases	Store JSON-like documents

Atlas manages infrastructure, security, backups and scaling so you can focus on building your apps.

ATLAS ARCHITECTURE



- Data is automatically replicated across availability zones.
- Choose the best region for low latency.
- Automatic failover ensures high availability.
- Scale up or down with just a few clicks.

CLUSTER TIERS

FREE TIER	SHARED	DEDICATED	MULTI-CLOUD	SERVERLESS
512 MB Storage Shared Clusters For learning & testing	For small apps Cost-effective	For production apps More power & control	Deploy on AWS, Azure or GCP	Auto scale Pay per request

Pay only for what you use. Scale storage & compute as your application grows.

CONNECT TO ATLAS

1. Get Connection String

Go to Cluster → Connect → Connect your application.

Choose your driver and copy the connection string.

2. Example (Node.js)

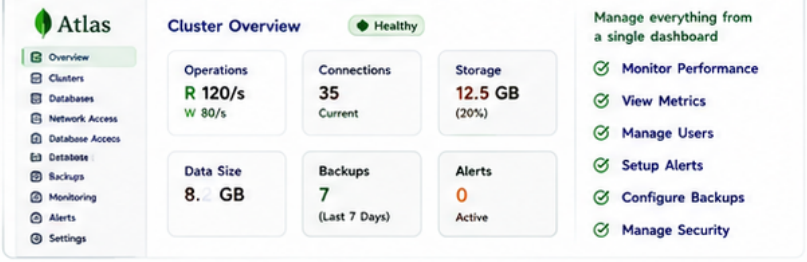
```
const { MongoClient } = require('mongodb');

const uri =
  "mongodb+srv://<username>:<password>@cluster0.xxxxx.mongodb.net/myDB?retryWrites=true&w=majority";

const client = new MongoClient(uri);
 await client.connect();
 console.log("Connected to Atlas!");
```

Atlas uses TLS encryption, IP Access Lists and Database Users to keep your data secure.

ATLAS DASHBOARD OVERVIEW



Manage everything from a single dashboard

- Monitor Performance
- View Metrics
- Manage Users
- Setup Alerts
- Configure Backups
- Manage Security

SECURITY FEATURES

- **IP Access List** — Allow connections from specific IPs.
- **Database Users** — Create users with specific roles.
- **TLS/SSL Encryption** — Encrypts data in transit.
- **Private Endpoints** — Secure access within your cloud.
- **Audit Logging** — Track and monitor all activities.

USE CASES

- **Web & Mobile Applications** — Reliable and scalable backend.
- **IoT & Real-time Analytics** — Store and process large amounts of data.
- **E-commerce Platforms** — Handle product, users, orders and more.
- **SaaS Applications** — Multi-tenant and highly available.
- **Global Applications** — Deploy closer to your users worldwide.

MongoDB Atlas = Cloud Database + Simplicity + Security + Scalability

MongoDB Compass

The Official GUI for MongoDB — Explore, Visualize & Manage Your Data

WHAT IS MONGODB COMPASS?

MongoDB Compass is the official GUI (Graphical User Interface) to connect, explore and manage your MongoDB databases easily.

KEY BENEFITS

- Beautiful & Intuitive UI
- Explore Data Easily
- Build Queries Visually
- Run Aggregations
- Analyze Schema
- Manage Indexes
- Improves Productivity
- Free and Open Source

QUICK FACTS

- Supports MongoDB Atlas and Local Databases
- Works on Windows, macOS and Linux
- Secure Connection (SSL)
- Built by MongoDB
- Free to Use

PRO TIP

Use Compass to explore your data, understand schema and optimize performance faster!

HOW MONGODB COMPASS WORKS

Connect	Databases	Collections	Documents	Queries	Aggregation	Indexes	Schema
Connect to your MongoDB instance	View all databases in your instance	Browse collections inside databases	View, add, edit and delete documents	Build and run queries easily	Build aggregation pipelines	View and manage indexes	Analyze schema and data types

DOCUMENTS VIEW (BROWSE & EDIT DATA)

COMPASS

My Cluster
Databases
Performance
Schema
Aggregations
Indexes
Settings

Databases

Search databases...

- admin
- config
- ecommerce
  - collections
    - products
    - users
    - orders
- local

ecommerce.products

Documents | Aggregations | Schema | Indexes

ADD DATA | VIEW | FILTER | OPTIONS

Displaying documents 1 - 5 of 5

```
_id: ObjectId("664b7a3e8f1a2b3c4d5e6f01")
name: "Wireless Headphones",
category: "Electronics",
price: 199.99
stock: 45
tags: [ "audio", "bluetooth", "wireless" ]
createdAt: 2024-05-20T10:15:30.000+00:00
```

```
_id: ObjectId("664b7a3e8f1a2b3c4d5e6f02")
name: "Smart Watch",
category: "Wearables",
price: 149.99
stock: 30
tags: [ "watch", "fitness", "bluetooth" ]
createdAt: 2024-05-20T10:16:45.000+00:00
```

SCHEMA ANALYSIS

Schema for ecommerce.products

Field	Type	% of Documents
_id	ObjectId	100%
name	String	100%
category	String	100%
price	Double	100%
stock	Int32	100%
tags	Array	100%
createdAt	Date	100%

Document Preview

```
{
  _id: ObjectId("664b7a3e8f1a2b3c4d5e6f01"),
  name: "Wireless Headphones",
  category: "Electronics",
  price: 199.99,
  stock: 45,
  tags: [ "audio", "bluetooth", "wireless" ],
  createdAt: ISODate("2024-05-20T10:15:30Z")
}
```

QUERIES (FIND DATA)

Filter {} { category: "Electronics" }

RESET | FIND

Query Results (2)

```
{ _id: ObjectId("...1"),
  name: "Wireless Headphones",
  category: "Electronics", ... }
```

```
{ _id: ObjectId("...3"),
  name: "Bluetooth Speaker",
  category: "Electronics", ... }
```

Execution Time: 3 ms

AGGREGATION PIPELINE

```
[
  { $match: { category: "Electronics" } },
  { $group: { _id: "$category",
              totalProducts: { $sum: 1 },
              avgPrice: { $avg: "$price" }
  } },
  { $sort: { avgPrice: -1 } }
]
```

INDEXES MANAGEMENT

CREATE INDEX | REFRESH

Name	Keys	Type	Unique	Sparse	Size	Usage
_id_	_id: 1	REGULAR	Yes	No	4.0 KB	High
category_1	category: 1	REGULAR	No	No	8.0 KB	Medium
price_1	price: 1	REGULAR	No	No	8.0 KB	Medium

CONNECTION (EXAMPLE)

Connection String (MongoDB Atlas)

```
mongodb+srv://<username>:<password>@cluster0.xxxxx
mongodb.net/?retryWrites=true&w=majority
```

Copy

Connection Status

● Connected Successfully

Cluster Type: Replica Set Version: 6.0.8 SSL: Enabled

USEFUL FEATURES

- Visual Query Builder
- Import / Export Data
- Schema Visualization
- Performance Insights
- Favorite Queries
- Dark / Light Theme

SHORTCUTS

⌘ N	New Query Tab
⌘ R	Refresh
⌘ B	Build Aggregation
⌘ F	Find in Documents
⌘ /	Show Shortcuts

MongoDB Compass = Connect + Explore + Visualize + Optimize

mongosh

The Official MongoDB Shell (Next Generation)

Interactive • Powerful • Developer Friendly

WHAT IS MONGOSH?

mongosh is the official MongoDB Shell.
It provides a modern, interactive way to run commands, query data and manage your MongoDB databases.

KEY FEATURES

- Modern & Interactive Shell
- Auto-completion
- Syntax Highlighting
- Multi-line Editing
- Built-in Helper Methods
- JSON Output Formatting
- Works with MongoDB Atlas & Local Instances

QUICK START

1. Install MongoDB or MongoDB Shell (mongosh)
2. Open Terminal / Command Prompt
3. Run: mongosh
4. Connect to your MongoDB instance

PRO TIP

Use Tab for auto-complete and Up/Down arrows to navigate command history.

mongosh

```
$ mongosh
Current Mongosh Log ID: 664b9f7c2e1ab1f8d1c0a123
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
Using MongoDB:          7.0.11
Using Mongosh:          2.2.6

test> show dbs
admin      40.00 KiB
config     12.00 KiB
local      80.00 KiB
test        0.00 B

test> use mydb
switched to db mydb

mydb> show collections
users
products
orders

mydb> db.users.find().limit(2).pretty()
  {
    _id: ObjectId('664b9f7c2e1ab1f8d1c0a001'),
    name: 'Alice',
    email: 'alice@example.com',
    age: 25
  }
  {
    _id: ObjectId('664b9f7c2e1ab1f8d1c0a002'),
    name: 'Bob',
    email: 'bob@example.com',
    age: 30
  }
mydb>
```

show dbs
List all databases

use database
Switch to a database

show collections
List collections in current database

db.collection.find()
Query documents from a collection

COMMON COMMANDS CHEAT SHEET

Command	Example	Description
show dbs	show dbs	List all databases
use <database>	use mydb	Switch to a database
show collections	show collections	List collections in current database
db.collection.find()	db.users.find()	Find all documents
db.collection.find().pretty()	db.users.find().pretty()	Find all documents (pretty JSON)
db.collection.find({})	db.users.find({ age: 25 })	Find documents with condition
db.collection.insertOne()	db.users.insertOne({ name:"John" })	Insert a single document
db.collection.updateOne()	db.users.updateOne({ name:"John" }, { $set: { age: 26 } })	Update one document
db.collection.deleteOne()	db.users.deleteOne({ name:"John" })	Delete one document
db.collection.aggregate()	db.orders.aggregate([{ $match: { status: "A" } }])	Run aggregation pipeline
db.collection.countDocuments()	db.users.countDocuments()	Count documents

MONGOSH SHORTCUTS

↑	Previous command
↓	Next command
Tab	Autocomplete
Ctrl + L	Clear screen
Ctrl + C	Cancel current operation
Ctrl + D	Exit mongosh

OUTPUT FORMATS

{ }	MongoDB Extended JSON (default)
{!}	Pretty JSON (use .pretty())
</>	Raw JSON (less formatting)

QUERY EXAMPLES

```
# All documents
db.users.find().pretty()

# Filter documents
db.users.find({ age: { $gt: 20 } })

# Projection (select fields)
db.users.find({}, { name: 1, email: 1, _id: 0 })

# Sort documents
db.users.find().sort({ age: -1 })

# Limit results
db.users.find().limit(5)
```

AGGREGATION EXAMPLE

```
db.orders.aggregate([
  { $match: { status: "A" } },
  { $group: {
      _id: "$customerId",
      total: { $sum: "$amount" }
    }},
  { $sort: { total: -1 } }
])
```

HELP & UTILITY

```
help                # Show help
help <command>      # Help for specific command
it                  # Repeat last command
cls / clear         # Clear screen
exit                # Exit mongosh
```

Type Tab to autocomplete commands and collection names.

CONNECTION EXAMPLES

```
# Local MongoDB
mongosh

# Local with custom port
mongosh --port 27018

# MongoDB Atlas
mongosh "mongodb+srv://<cluster0.xxxxx.mongodb.net/mydb" --apiVersion 1 --username <username>

# With authentication
mongosh -u <username> -p <password> --authenticationDatabase admin
```

mongosh = Modern Shell + Powerful Commands + Faster Development

Connection String

Connect Your Application to MongoDB

WHAT IS CONNECTION STRING?

A Connection String is a URL that provides all the information needed to connect your application to a MongoDB database.

KEY COMPONENTS

- **Username** — User to authenticate
- **Password** — User password
- **Host** — Database host or cluster address
- **Port** — Port number (default 27017)
- **Database** — Target database name
- **Connection Options** — Additional settings (optional)

COMMON SCHEMES

`mongodb://`	Standard Connection (For self-hosted MongoDB)
`mongodb+srv://`	DNS Seedlist Connection (For MongoDB Atlas)

PRO TIP

Use environment variables to store sensitive information like username and password. Never hardcode them in your code!

CONNECTION STRING ANATOMY

```
mongodb://username:password@host:port/database?options
```

SCHEME	CREDENTIALS	HOST AND PORT	DATABASE	OPTIONS
Protocol used to connect to MongoDB	Username and password for authentication	MongoDB server address and port number	Mame of the database to connect	Additional connection parameters

BUILD YOUR CONNECTION STRING

1. **Scheme** — `mongodb://` or `mongodb+srv://`
2. **Credentials** — `username:password` (If authentication is enabled)
3. **Host & Port** — `host:port` (Default port is 27017)
4. **Database** — `/databaseName` (Database to connect)
5. **Options** — `?key=value&key2=value2` (Optional parameters)

= `mongodb://username:password@host:port/database?options`

BEST PRACTICES

- **Use authentication** — Always use username and password.
- **Use mongodb+srv://** — Recommended for MongoDB Atlas.
- **Keep credentials safe** — Store in environment variables or secrets manager.
- **Use options wisely** — Add only necessary options for better performance.
- **Test your connection** — Always test connection string before deploying your application.

EXAMPLES

1 Standard Connection (Self-Hosted)

```
mongodb://admin:myPassword123@localhost:27017/mydb
```

Username · Password · Host · Port · Database

2 MongoDB Atlas Connection (Recommended)

```
mongodb+srv://user:pass123@cluster0.xxxxx.mongodb.net/mydb?retryWrites=true&w=majority
```

Username · Password · Cluster Host (Atlas) · Database · Options

For Atlas, the default port (27017) is used automatically with mongodb+srv://

COMMON CONNECTION OPTIONS

Option	Description	Example
retryWrites	Retry write operations	`retryWrites=true`
w	Write concern	`w=majority`
authSource	Authentication database	`authSource=admin`
ssl	Use SSL/TLS	`ssl=true`
connectTimeoutMS	Connection timeout (ms)	`connectTimeoutMS=5000`
serverSelectionTimeoutMS	Server selection Timeout (ms)	`serverSelectionTimeoutMS=5000`

SECURITY NOTE — Never share your connection string publicly. It contains sensitive information!

NODE.JS EXAMPLE

```
const { MongoClient } = require('mongodb');
const uri = "mongodb+srv://user:pass123@cluster0.xxxxx.mongodb.net/mydb?retryWrites=true&w=majority";
const client = new MongoClient(uri);
async function connect() {
  try {
    await client.connect();
    console.log('✅ Connected to MongoDB!');
  } catch (err) {
    console.error('❌ Connection failed:', err);
  }
}
connect();
```

A Good Connection String = Secure + Reliable + Fast Connection to Your MongoDB Database!

MongoDB Node.js Driver

Use MongoDB in Your Node.js Applications

WHAT IS MONGODB NODE.JS DRIVER?

The official MongoDB driver for Node.js allows you to connect to MongoDB and perform CRUD operations easily and efficiently.

KEY FEATURES

- Official & Actively Maintained
- High Performance
- Promise Based API
- Connection Pooling
- Support for Transactions
- Wide Platform Support
- TypeScript Friendly

INSTALLATION

```
> npm install mongodb
```

QUICK START

```
import { MongoClient } from 'mongodb';

const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);

await client.connect();
console.log("Connected to MongoDB");
await client.close();
```

PRO TIP

Reuse the same MongoClient instance throughout your application for better performance (connection pooling).

NODE.JS DRIVER WORKFLOW

All database operations are asynchoncous and return Promises.

NODE.JS DRIVER API OVERVIEW

Method	Description	Example	Returns
MongoClient	Creates a new client instance	`const client = new MongoClient(uri);`	MongoClient
connect()	Connects to MongoDB server	`await client.connect();`	Promise<void>
db()	Gets a database instance	`const db = client.db("mydb");`	Db
collection()	Gets a collection instance	`const col = db.collection("users");`	Collection
find()	Finds documents	`await col.find({ age: { $gt: 18 } }).toArray();`	Cursor / Array
insertOne()	Inserts a single document	`await col.insertOne({ name: "John", age: 25 });`	InsertOneResult
updateOne()	Updates a single document	`await col.updateOne( { name: "John" }, { $set: { age: 26 } } );`	UpdateResult
deleteOne()	Deletes a single document	`await col.deleteOne({ name: "John" });`	DeleteResult

COMMON OPERATIONS EXAMPLES

1 Connect to MongoDB

```
import { MongoClient } from 'mongodb';
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);
await client.connect();                          // Connect to server
```

2 Get Database & Collection

```
const db = client.db('mydb');                    // Select database
const users = db.collection('users');            // Select collection
```

3 Find Documents

```
const cursor = users.find({ age: { $gt: 18 } });   // Query
const result = await cursor.toArray();             // Get results
cnsoole.log(result);
```

4 Insert One Document

```
const doc = { name: "Alice", age: 22 };
const res = aweit users.insertOne(doc);          // Insert
onssole.log("Inserted ID:", res.insertedId);
```

5 Update One Document

```
const res = aweit users.updateOne(               // Update
   { name: "Alice^ },
   { $set: { age: 23 } }
);
consolle.log("Matched'"; res.matchedCount, /)
             "Modified'"; res.modifiedCount);
```

6 Delete One Document

```
const res = aweit users.deleteOne({ name: "Alice" });
console.log("Deleted:", res.deletedCount);       // Delete
```

7 Close Connection

```
aweit client.close();
console.log("Connection closed");                // Close
```

BEST PRACTICES

- Use a single MongoClient instance (connection pooling).
- Always handle errors with try/catch.
- Use async/await for clean code.
- Create indexes for better performance.
- Store credentials securely using environment variables.
- Close the connection when the application exits.

MongoDB + Node.js Driver
Powerful Combination for Modern Applications!

Mongoose

Elegant MongoDB Object Modeling for Node.js

WHAT IS MONGOOSE?

Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js. It provides a schema-based solution to model your data.

KEY FEATURES

- Schema-Based Modeling
- Built-in Validation
- Middleware (Pre & Post Hooks)
- Powerful Query API
- Population (Relationships)
- Easy Data Casting
- Indexing & Virtuals
- Lean Queries for Performance

MONGOOSE WORKFLOW

Define Schema → Create Model → Work with Documents → Stored in MongoDB

QUICK START

```
$ npm install mongoose
```

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://127.0.0.1:27017/mydb')
  .then(() => console.log('Connected!'))
  .catch(err => console.error(err));
```

PRO TIP

Use .lean() for read-only operations when you don't need full Mongoose documents. It improves performance!

MONGOOSE ARCHITECTURE

Schema — Define structure and rules → **Model** — Compiles schema into a Model → **Document** — Create, read, update and delete data → **Queries** — Powerful & chainable query builder → **MongoDB** — Data stored in MongoDB

Mongoose sits on top of the native MongoDB driver and adds a layer of abstraction.

CORE CONCEPTS

Schema

Defines the structure, fields, types, defaults, validation and more.

```
const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, required: true },
  age: Number
});
```

Model

A Model is a constructor compiled from a Schema.

```
const User = mongoose.model('User', userSchema);
```

Document

An instance of a Model. Represents a record in a collection.

```
const user = new User({
  name: 'John',
  email: 'john@example.com',
  age: 25
});
```

Queries

Mongoose provides powerful query helpers.

```
User.find({ age: { $gt: 18 }})
  .sort('-createdAt')
  .limit(10)
  .exec((err, users) => {
    // handle result
  });
```

MIDDLEWARE (HOOKS)

Pre Hooks (Before)

```
userSchema.pre('save', function(next) {
  console.log('Before saving user...');
  next();
});
```

Post Hooks (After)

```
userSchema.post('save', function(doc) {
  console.log('User saved:', doc._id);
});
```

Mongoose Middleware is executed in sequence and is great for

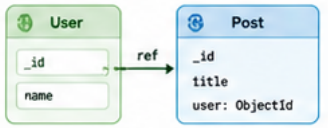
Validation · Logging · Modify Data · Business Logic

VALIDATION

```
const userSchema = new mongoose.Schema({
  email: { type: String, required: [true, 'Email is required'] },
  age:  { type: Number, min: [18, 'Must be at least 18'] },
  name:  { type: String, maxlength: [50, 'Too long!'] }
});
```

Built-in validators ensure data integrity and clean records.

POPULATE (RELATIONSHIPS)




```
const User = mongoose.model('User', userSchema);
const Post = mongoose.model('Post', postSchema);

Post.find().populate('user').exec()
  .then(posts => console.log(posts));
```

Populate lets you replace references (ObjectId) with actual documents.

COMMON QUERIES

Operation	Example	Description
Find	`User.find({ age: { $gt: 18 }})`	Find documents
Find One	`User.findOne({ email: 'a@b.com' })`	Find single document
Insert	`User.create({ name: 'John' })`	Insert a new document
Update	`User.updateOne({ _id: id }, { $set: { age: 26 }})`	Update one document
Delete	`User.deleteOne({ _id: id })`	Delete one document
Count	`User.countDocuments({ active: true })`	Count documents

EXAMPLE: COMPLETE FLOW

1. Connect

```
mongoose.connect('mongodb://127.0.0.1:27017/mydb');
```

2. Define Schema

```
const userSchema = new mongoose.Schema({
  name: String, email: String, age: Number
});
```

3. Create Model

```
const User = mongoose.model('User', userSchema);
```

4. Create Document

```
const user = new User({ name: 'John',
  email: 'john@example.com', age: 25 });
await user.save();
```

5. Query Documents

```
const users = await User.find({ age: { $gt: 18 }})
  .sort('-age').exec();
```

6. Update Document

```
await User.updateOne({ _id: user._id },
  { $set: { age: 26 } });
```

7. Delete Document

```
await User.deleteOne({ _id: user._id });
```

Mongoose makes working with MongoDB in Node.js easier, cleaner and more powerful!

Mongoose Schema

Define Structure, Types & Rules for Your Data

WHAT IS MONGOOSE SCHEMA?

A Mongoose Schema defines the structure of documents in a collection. It includes field names, data types, validation rules, defaults and more.

KEY FEATURES

- Define Fields & Data Types
- Built-in Validation
- Default Values
- Required Fields
- Nested Objects & Arrays
- References (ObjectId)
- Custom Validators
- Middleware Support

MONGOOSE FLOW

Schema (Structure) → Model (Compiled) → Collection (MongoDB) → Documents (JSON/BSON)

PRO TIP

Define a good schema to enforce data consistency and make your application more reliable & maintainable.

SCHEMA ANATOMY

```
const schema = new mongoose.Schema({ /* fields */ }, { /* options */ });
```

- Fields (Structure)
- Schema Options
- Returned Schema Object

COMMON DATA TYPES IN MONGOOSE SCHEMA

Type	Description	Example	JS Type	BSON Type
String	Text data	`name: { type: String }`	String	String
Number	Numeric values	`age: { type: Number }`	Number	Int32 / Double
Boolean	True / False	`isActive: { type: Boolean }`	Boolean	Boolean
Date	Date & time	`createdAt: { type: Date }`	Date	Date
Array	List of values	`tags: { type: [String] }`	Array	Array
Object	Nested object	`address: { type: Object }`	Object	Document
ObjectId	Reference to another doc	`user: { type: ObjectId, ref: 'User' }`	ObjectId	ObjectId

SCHEMA OPTIONS (SECOND PARAMETER)

Option	Description	Example
timestamps	Adds createdAt & updatedAt	`{ timestamps: true }`
collection	Custom collection name	`{ collection: 'users' }`
strict	Disallow fields not in schema	`{ strict: true }`
toJSON	Transform output to JSON	`{ toJSON: { virtuals: true } }`

FIELD OPTIONS (VALIDATION & BEHAVIOR)

Option	Description	Example
required	Field is required	`{ type: String, required: true }`
default	Default value if not provided	`{ type: String, default: 'N/A' }`
unique	Must be unique	`{ type: String, unique: true }`
enum	Value must be from list	`{ type: String, enum: ['M', 'F'] }`
min / max	Minimum / Maximum value	`{ type: Number, min: 0, max: 120 }`
match	Match regex pattern	`{ type: String, match: /.+@.+\..+/ }`
trim	Remove spaces from string	`{ type: String, trim: true }`

COMPLETE SCHEMA EXAMPLE

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  email: { type: String, required: true, unique: true, match: /.+@.+\..+/ },
  age: { type: Number, min: 18, max: 100, default: 18 },
  isActive: { type: Boolean, default: true },
  role: { type: String, enum: ['User', 'Admin'], default: 'User' },
  tags: { type: [String], default: [] },
  address: {
    street: { type: String, required: true },
    city: { type: String, required: true },
    zip: { type: String }
  },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now },
 profilePicture: { type: mongoose.Schema.Types.ObjectId, ref: 'Profile' }
},{
  timestamps: true,
  collection: 'users'
});
const User = mongoose.model('User', userSchema);
```

Includes:
- Different Data Types
- Validation Rules
- Defaults
- Enums
- Arrays
- Nested Objects
- References
- Timestamps

A Well-Defined Schema = Clean Data + Validation + Better Performance + Easy Maintenance

Mongoose Model

Models are Compiled from Schemas to Create & Work with Documents

WHAT IS A MODEL?

A Model is a wrapper around a Schema.
It provides an interface to the database for creating, querying, updating and deleting documents.

KEY FEATURES

- Compiled from Schema
- Create & Read Documents
- Update & Delete Documents
- Powerful Query API
- Middleware (Hooks) Support
- Validation & Defaults
- Works with Collections

MODEL FLOW

Schema (Structure) → Model (Compiled) → Collection (MongoDB) → Document (Data)

PRO TIP

Use Models to interact with MongoDB in a clean, simple and powerful way!

HOW MODEL WORKS

1. **Schema** — Define structure & rules
2. **Model** — Compile Schema into Model
3. **Collection** — Model maps to a MongoDB collection
4. **Document** — Create, Read, Update, Delete Documents
5. **MongoDB** — Data is stored in MongoDB

Each Model is linked to a single Collection in MongoDB.

COMMON MODEL METHODS

Method	Description	Returns	Example
create()	Insert one or more new documents	Document / Array of Documents	`User.create({ name: 'John', age: 25 })`
find()	Find all documents that match the filter	Array of Documents	`User.find({ isActive: true })`
findOne()	Find the first document that matches the filter	Document	`User.findOne({ email: 'john@example.com' })`
findById()	Find document by its ObjectId	Document	`User.findById('665f3c2a1e8a2b2a9d5c1234')`
updateOne()	Update the first document that matches the filter	UpdateResult	`User.updateOne({ name: 'John' }, { $set: { age: 26 } })`
deleteOne()	Delete the first document that matches the filter	DeleteResult	`User.deleteOne({ name: 'John' })`

CREATE MODEL FROM SCHEMA

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, unique: true },
  age: { type: Number, default: 18 },
  isActive: { type: Boolean, default: true },
  createdAt: { type: Date, default: Date.now }
}, { timestamps: true });

// Create Model (compiles schema into a model)
const User = mongoose.model('User', userSchema);
// Collection name will be "users" (plural, lowercase)
module.exports = User;
```

Schema (Structure & Rules) → Model (Interface) → Collection (users)

MODEL METHODS EXAMPLES

```
// 1. create()
const user = await User.create({ name: 'Alice', email: 'alice@gmail.com', age: 22 });

// 2. find()
const users = await User.find({ isActive: true });

// 3. findOne()
const user = await User.findOne({ email: 'alice@gmail.com' });

// 4. findById()
const user = await User.findById('665f3c2a1e8a2b2a9d5c1234');

// 5. updateOne()
const result = await User.updateOne(
    { email: 'alice@gmail.com' },
    { $set: { age: 23 } }
  );

// 6. deleteOne()
const del = await User.deleteOne({ email: 'alice@gmail.com' });
```

Document Example

```
{
  _id: ObjectId("665f3c2a1e8a2b2a9d5c1234"),
  name: "Alice",
  email: "alice@gmail.com",
  age: 22,
  isActive: true,
  createdAt: ISODate("2024-05-20T10:15:30Z"),
  updatedAt: ISODate("2024-05-20T10:15:30Z")
}
```

MODEL vs COLLECTION

Model	Collection
✓ Created from Schema	✓ Actual storage in MongoDB
✓ Provides methods to work with data	✓ Stores documents
✓ High-level abstraction	✓ Low-level database object
✓ Example: User (Model)	✓ Example: users (Collection)

VS

BEST PRACTICES

- Always define a schema before creating a model.
- Use validation, defaults and required fields.
- Use indexes for better query performance.
- Use middleware (hooks) for common tasks.
- Handle errors properly in async operations.

Models make your code clean, organized and powerful! Build better applications with Mongoose.

Mongoose CRUD Operations

Create, Read, Update & Delete Data with Mongoose

WHAT IS CRUD?

CRUD stands for Create, Read, Update & Delete. Mongoose provides powerful methods to perform CRUD operations on MongoDB collections.

MONGOOSE FLOW

Schema (Structure) → Model (Compiled) → Collection (MongoDB) → Document (Data)

KEY FEATURES

- Easy to Use Methods
- Schema-Based Validation
- Powerful Query API
- Middleware (Hooks) Support
- Works with Promises / async-await
- Clean & Readable Code

PRO TIP

Use async/await with try/catch for better error handling and cleaner code.

CREATE

Add new document(s) to the collection.

`create()`

CREATE - Insert One

```
const User = mongoose.model('User', userSchema);
const newUser = await User.create({
  name: 'John',
  email: 'john@example.com',
  age: 25,
  isActive: true
});
console.log('User Created:', newUser);
```

RESULT (Created Document)

```
{
  _id: ObjectId("665f3c2a1e8a2b2a9d5c1234"),
  name: "John",
  email: "john@example.com",
  age: 25,
  isActive: true,
  __v: 0
}
```

READ

Retrieve document(s) from the collection.

`find()`

READ - Find Many

```
const users = await User.find({ isActive: true })
  .sort({ age: -1 })
  .limit(10);

console.log('Users:', users);
```

RESULT (Documents Found)

```
[
  { _id: ObjectId("665f3c2a1e8a..."),
  name: "John", email: "john@example.com",
  age: 25, isActive: true },
  { _id: ObjectId("665f3c2a1e8a..."),
  name: "Alice", email: "alice@example.com",
  age: 30, isActive: true }
]
```

UPDATE

Modify existing document(s).

`updateOne()`

UPDATE - Update One

```
const result = await User.updateOne(
  { email: 'john@example.com' },
  { $set: { age: 26, isActive: false } }
);
console.log('Matched:', result.matchedCount);
console.log('Modified:', result.modifiedCount);
```

RESULT (Update Result)

```
{
  "acknowledged": true,
  "matchedCount": 1,
  "modifiedCount": 1
}
```

DELETE

Remove document(s) from the collection.

`deleteOne()`

DELETE - Delete One

```
const result = await User.deleteOne(
  { email: 'john@example.com' }
);
console.log('Deleted:', result.deletedCount);
```

RESULT (Delete Result)

```
{
  "acknowledged": true,
  "deletedCount": 1
}
```

MONGOOSE CRUD METHODS OVERVIEW

Operation	Method	Description		Example
Create	create()	Insert one or more documents	Document / Array	User.create({ name: 'John' })
Read	find() / findOne()	Find documents	Array / Document	User.find({ age: { $gt: 18 } })
Update	updateOne() / updateMany()	Update document(s)	UpdateResult	User.updateOne({ _id: id }, { $set: { age: 26 } })
Delete	deleteOne() / deleteMany()	Delete document(s)	DeleteResult	User.deleteOne({ _id: id })

BEST PRACTICES

- Always use async/await for better readability.
- Handle errors using try/catch.
- Validate data using Schema before saving.
- Use indexes for better query performance.

COMMON QUERY OPERATORS

$eq	: Equal to	$gte	: Greater than or equal
$ne	: Not equal to	$lte	: Less than or equal
$gt	: Greater than	$in	: In array
$lt	: Less than	$nin	: Not in array

CRUD Operations are the foundation of any application. Mongoose makes it simple, powerful & developer-friendly!

Mongoose Population

populate() से References को Related Data में बदलें

WHAT IS POPULATE?

Mongoose का populate() method ObjectId references को उनके असली documents के साथ replace करता है। यह related data को आसानी से fetch करने देता है।

MONGOOSE FLOW

Document with ObjectId → populate() Method → Document with Related Data

KEY FEATURES

- References को आसान तरीके से fetch करें
- Multiple levels तक populate कर सकते हैं
- Query के साथ filter और select कर सकते हैं
- Performance बेहतर, Code साफ और आसान
- One-to-One और One-to-Many relations

SAMPLE DOCUMENTS

User Collection

```
{
  "_id": ObjectId("64f8e6b2..."),
  "name": "John Doe",
  "email": "john@example.com"
}
```

Post Collection

```
{
  "_id": ObjectId("64f8e7a1..."),
  "title": "My First Post",
  "content": "Hello World!",
  "author": ObjectId("64f8e6b2...")
}
```

Post.author → User._id (ObjectId Reference)

EXAMPLE: USER & POSTS RELATIONSHIP

User Schema

```
const userSchema = new Schema({
  name: String,
  email: String
});

const User = model('User', userSchema);
```

User ⟷ Posts (One-to-Many Relationship)

Post Schema

```
const postSchema = new Schema({
  title: String,
  content: String,
  author: {
    type: Schema.Types.ObjectId,
    ref: 'User'
  }
});
const Post = model('Post', postSchema);
```

Post में author field User का ObjectId reference है (ref: 'User')

BEFORE & AFTER POPULATION

BEFORE populate() (Contains ObjectId)

```
{
  "_id": ObjectId("64f8e7a1..."),
  "title": "My First Post",
  "content": "Hello World!",
  "author": ObjectId("64f8e6b2...")
}
```

AFTER populate() (Contains Related Data)

```
{
  "_id": ObjectId("64f8e7a1..."),
  "title": "My First Post",
  "content": "Hello World!",
  "author": {
    "_id": ObjectId("64f8e6b2..."),
    "name": "John Doe",
    "email": "john@example.com"
  }
}
```

★ populate() ObjectId को पूरी Document में बदल देता है।

POPULATE() EXAMPLES

1. Basic populate()

```
// All posts with user details
Post.find()
  .populate('author')
  .exec((err, posts) => {
   console.log(posts);
} });
```

2. Select specific fields

```
// Only name and email
Post.find()
  .populate('author', 'name email')
  .exec((err, posts) => {
   console.log(posts);
} });
```

3. Populate with match condition

```
// Only active users
Post.find()
  .populate({
    path: 'author',
    match: { isActive: true }
  })
  .exec((err, posts) => {
   console.log(posts);
  });
```

4. Populate multiple paths

```
// Populating multiple references
Comment.find()
  .populate('post')    // Post details
  .populate('author')  // User details
  .exec((err, comments) => {
   console.log(comments);
} });
```

5. Nested/Deep populate

```
// Populate friends of user
Post.find()
  .populate({
     path: 'author',
     populate: { path: 'friends', select: 'name email' }
  })
  .exec((err, posts) => {
   console.log(posts);
} });
```

HOW POPULATE() WORKS?

1. Query execute होती है और documents मिलते हैं
2. ObjectId references को collect किया जाता है
3. Referenced collections से related documents fetch होते हैं
4. Original documents में related data replace करके return किया जाता है

BEST PRACTICES

- Only necessary fields populate करें (select का use करें).
- Large data होने पर limit और match का use करें.
- Deep populate का use सोच-समझकर करें.
- Indexes का use करें better performance के लिए.

Mongoose populate() आपके data relationships को आसान और powerful बनाता है! ★

MongoDB Transactions

Perform Atomic Operations Across Multiple Documents & Collections

WHAT IS A TRANSACTION?

A transaction groups multiple operations into a single unit of work. Either all operations succeed (commit) or all fail (rollback). Ensures data consistency and reliability.

KEY FEATURES

- Atomicity – All or nothing
- Consistency – Data remains valid
- Isolation – Transactions are isolated
- Durability – Committed data persists
- Support for Multiple Collections
- Rollback on Error
- ACID Compliance

TRANSACTION FLOW

Start Session → Start Transaction → Operations (Reads/Writes) → Commit or Abort

PRO TIP

Use transactions only when necessary. They have some performance cost.

HOW TRANSACTIONS WORK

Session	startTransaction()	Operations	commitTransaction()	abortTransaction()
Create a session connection	Start the transaction	Perform multiple read/write operations	Commit if all operations succeed	Abort & rollback if any operation fails

All operations in a transaction are executed atomically.

TRANSACTION METHODS

Method	Description	Example
startSession()	Create a new session	`const session = client.startSession();`
startTransaction()	Start a transaction on the session	`session.startTransaction();`
commitTransaction()	Commit the transaction (make changes permanent)	`await session.commitTransaction();`
abortTransaction()	Abort the transaction (rollback all changes)	`await session.abortTransaction();`
endSession()	End the session	`session.endSession();`

ACID PROPERTIES

A	Atomicity	All operations in a transaction succeed or none at all.
C	Consistency	Database remains in a valid state before and after transaction.
I	Isolation	Transactions are isolated from each other.
D	Durability	Once committed, changes are permanent.

ROLLBACK EXAMPLE

```
try {
   session.startTransaction();
   // Some operations...
   throw new Error('Something went wrong');
   await session.commitTransaction();
} catch (err) {
   await session.abortTransaction();
   console.log('All changes rolled back');
} finally {
   session.endSession();
}
```

All operations are undone.

TRANSACTION EXAMPLE (NODE.JS DRIVER)

```
const { MongoClient } = require('mongodb');

async function transferMoney() {
  const client = new MongoClient('mongodb://localhost:27017');
  const session = client.startSession();

  try {
    session.startTransaction();

    // 1. Deduct from sender
    await client.db('bank').collection('accounts').updateOne(
      { _id: 1 },
      { $inc: { balance: -100 } },
      { session }
    );

    // 2. Add to receiver
    await client.db('bank').collection('accounts').updateOne(
      { _id: 2 },
      { $inc: { balance: 100 } },
      { session }
    );

    // Commit the transaction
    await sesiеn.commitTransaction();
    console.log('Transaction committed successfully');
  } catch (err) {
    // Rollback the transaction
    await session.abortTransaction();
    console.error('Transaction aborted:', err.message);
  } finally {
    await session.endSession();
    await client.close();
  }
}
transferMoney();
```

client.startSession() Session → startTransaction() Begin Transaction → Perform Operations (Read / Write) → commitTransaction() Success → abortTransaction() On Error (Rollback)

WHEN TO USE TRANSACTIONS?

- When you need Atomic operations across multiple documents/collections.
- When data consistency is critical.
- For financial operations (money transfer, payments, orders etc.).
- When multiple related writes must succeed together.

IMPORTANT NOTES

- Transactions require replica set or sharded cluster.
- Keep transactions short to avoid timeouts and locks.
- Transactions may impact performance if overused.
- Not supported on standalone MongoDB instances.

Example Use Case: Transfer Money

User A (Balance: 1000) → startTransaction() → Deduct 100 from A (Balance: 900) → Add 100 to B (Balance: 1100) → commitTransaction() → All changes saved atomically

Transactions in MongoDB help you write reliable, consistent and error-free applications.

MongoDB Sessions

Manage Client Connections & Enable Transactions

WHAT IS A SESSION?

A session is a logical connection between the client and MongoDB server. It enables causal consistency and supports transactions.

KEY FEATURES

- Logical connection to server
- Supports transactions
- Causal consistency
- Tracks operation order
- Automatic resource management
- Works across multiple operations

SESSION FLOW

PRO TIP

Always end your session with *endSession()* to free server resources.

HOW A SESSION WORKS

1. Client Session

Client requests a session from MongoDB server.

`startSession()`

2. Session Created

MongoDB creates a logical session with a unique Session ID.

Session ID
lsid: { id: UUID }

3. Use Session

All operations are executed in the context of this session.

Operations
(Read / Write)

4. End Session

Session is ended after commit or abort.

Commit (Success) | Abort (Failure)

Sessions provide context for operations and enable advanced features like transactions and causal consistency.

SESSION METHODS

Method	Description	Example
startSession()	Creates a new client session	`const session = client.startSession();`
withSession()	Runs operations with the given session	`client.withSession(session, async () => { /* operations */ });`
startTransaction()	Starts a transaction on the session	`session.startTransaction();`
commitTransaction()	Commits all operations in the transaction	`await session.commitTransaction();`
abortTransaction()	Aborts (rolls back) all operations	`await session.abortTransaction();`
endSession()	Ends the session and releases resources	`await session.endSession();`

SESSION INFORMATION

Session ID (lsid)

```
lsid: { id: UUID("a1b2c3d4-e5f6-7890-abcd-ef1234567890") }
uid: <optional user id>
clusterTime: <server timestamp>
operationTime: <timestamp>
```

WHEN TO USE SESSIONS?

- When using transactions
- When you need causal consistency
- When tracking sequence of operations
- For retryable writes

CODE EXAMPLE (Node.js Driver)

```
const { MongoClient } = require('mongodb');

async function run() {
  const client = new MongoClient('mongodb://localhost:27017');

  // 1. Start a session
  const session = client.startSession();
  console.log('Session started');
  console.log('Session ID:', session.id);

  try {
    // 2. Use session for operations or transactions
    session.startTransaction();

    await client.db('testdb').collection('users')
      .insertOne({ name: 'John', age: 25 }, { session });

    await client.db('testdb').collection('users')
      .updateOne({ name: 'John' }, { $set: { age: 26 } }, { session });

    // 3. Commit the transaction
    await session.commitTransaction();
    console.log('Transaction committed');
  } catch (err) {
    // 4. Abort on error
    await session.abortTransaction();
    console.error('Transaction aborted', err.message);
  } finally {
    // 5. End the session
    await session.endSession();
    await client.close();
    console.log('Session ended');
  }
}
run().catch(console.error);
```

SESSION LIFECYCLE

BENEFITS

Atomic Operations | Data Consistency | Rollback on Failure | Better Reliability & Control | Improved Performance

Sessions are the foundation for reliable operations and transactions in MongoDB.

MongoDB Atomic Operations

Small Changes, Big Guarantee – Atomic by Default

WHAT ARE ATOMIC OPERATIONS?

Atomic operations in MongoDB ensure that a write operation on a single document is completed fully or not at all. No intermediate state is visible to other operations.

KEY FEATURES

- Atomic at Single Document Level
- No Partial Updates
- Consistent Data
- Isolation from Other Operations
- High Performance
- Built-in Safety

PRO TIP

Use atomic operators ($set, $inc, $push, $pull) to update only the required fields instead of replacing the whole document.

ATOMIC UPDATE OPERATORS

$set

Sets the value of a field.

Example

```
{$set: { name: "John" }}
```

$inc

Increments the value of a field.

Example

```
{$inc: { age: 1 }}
```

$push

Adds an element to an array.

Example

```
{$push: { skills: "JS" }}
```

$pull

Removes matching elements from an array.

Example

```
{$pull: { skills: "PHP" }}
```

ATOMIC OPERATIONS WORK ON SINGLE DOCUMENT

BEFORE UPDATE

```
{
  "_id": 1,
  "name": "John",
  "age": 25,
  "skills": ["C++", "PHP"],
  "loginCount": 5
}
```

→ Atomic Operation
All or Nothing →

AFTER UPDATE ($inc: { age: 1 })

```
{
  "_id": 1,
  "name": "John",
  "age": 26,
  "skills": ["C++", "PHP"],
  "loginCount": 5
}
```

ATOMIC UPDATE EXAMPLES

```
// 1. $set - update a field
db.users.updateOne(
  { _id: 1 },
  { $set: { name: "John", city: "Delhi" } }
);

// 2. $inc - increment a field
db.users.updateOne(
  { _id: 1 },
  { $inc: { age: 1, loginCount: 1 } }
);

// 3. $push - add to array
db.users.updateOne(
  { _id: 1 },
  { $push: { skills: "JavaScript" } }
);

// 4. $pull - remove from array
db.users.updateOne(
  { _id: 1 },
  { $pull: { skills: "PHP" } }
);
```

SAMPLE DOCUMENT

```
{
  "_id": 1,
  "name": "John",
  "age": 25,
  "city": "Delhi",
  "skills": ["C++", "PHP"],
  "loginCount": 5
}
```

★ Each update above is atomic on a single document.

COMMON ATOMIC OPERATORS CHEAT SHEET

Operator	Use	Result
$set	Set the value of a field	{$set: { status: "active" }}
$inc	Increment the value	{$inc: { points: 10 }}
$push	Add an element to array	{$push: { tags: "new" }}
$pull	Remove matching element	{$pull: { tags: "old" }}

CONSISTENCY GUARANTEED

Atomic operations ensure:

- Data remains consistent
- No partial writes
- Predictable results
- Safe for concurrent access

MongoDB ensures your data is always consistent and safe.

SAFE UPDATE PRACTICES

- Always use atomic operators for field updates.
- Avoid replacing the whole document (unless necessary).
- Validate data before update.
- Use appropriate filters to target the right document.
- Keep operations small and focused.

Atomic Operations = Safe Updates + Consistency + Performance

MongoDB Change Streams

Real-Time Data Changes • Event Driven • Live Notifications

WHAT ARE CHANGE STREAMS?

Change Streams provide a way to watch for changes in your MongoDB data in real-time.

Get notified when documents are Inserted, Updated or Deleted.

KEY FEATURES

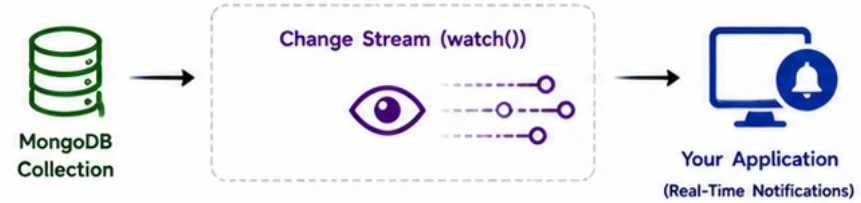
- Real-time data change notifications
- Watch specific collections or databases
- Detect Insert, Update, Delete operations
- Works across replica sets & sharded clusters
- Resume from last processed event
- Built-in in MongoDB (No extra tools)
- Great for live dashboards, logs, caches, and event-driven apps

PRO TIP

Use Change Streams to build real-time features without polling the database.

Efficient, fast and scalable!

HOW CHANGE STREAMS WORK



CHANGE EVENTS

INSERT

Triggered when a new document is inserted.

```
operationType: "insert"
fullDocument: { ... }
```

UPDATE

Triggered when a document is updated.

```
operationType: "update"
updateDescription: { ... }
```

DELETE

Triggered when a document is deleted.

```
operationType: "delete"
documentKey: { _id: ... }
```

OTHER EVENTS

Includes replace, invalidate, drop, etc.

```
operationType: "replace"
operationType: "invalidate"
```

SAMPLE CHANGE EVENT (INSERT)

```
{
  "_id": { "$oid": "665f6c1e8f1c2b3d4e5f6789" },
  "operationType": "insert",
  "fullDocument": {
    "_id": 101,
    "name": "Laptop",
    "price": 55000,
    "stock": 10
  },
  "ns": { "db": "test", "coll": "products" },
  "documentKey": { "_id": 101 },
  "clusterTime": { "$timestamp": { "t": 1717000001, "i": 1 } }
}
```

COMMON USE CASES

- Real-time Analytics Dashboards
- Instant Notifications & Alerts
- Cache Invalidation
- Audit Logs & Activity Tracking
- Integrations & Event Driven Systems

EXAMPLE (Node.js Driver)

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = new MongoClient('mongodb://localhost:27017');
  await client.connect();

  const db = client.db('test');
  const collection = db.collection('products');

  // Start watching the collection
  const changeStream = collection.watch();
  console.log('Watching for changes...');

  changeStream.on('change', (change) => {
    console.log('Change detected:');
    console.log(JSON.stringify(change, null, 2));
  });

  changeStream.on('error', (err) => {
    console.error('Change Stream Error:', err);
  });

  // Close connection on exit
  process.on('SIGINT', async () => {
    await changeStream.close();
    await client.close();
    process.exit(0);
  });
}
main().catch(console.error);
```

watch()
Start Change Stream

↓

Listen for Events
(Insert/Update/Delete)

↓

Process Change
in Real-Time

↓

Send Notifications /
Update UI / Logs

↓

Stop Stream
& Close

SUPPORT & REQUIREMENTS

- Replica Set Required
- Sharded Cluster Supported
- Not Available on Standalone Server

BEST PRACTICES

- Always handle errors and reconnect logic.
- Store resumeToken to continue stream from last event.
- Filter events using pipeline (match) to reduce noise.
- Close streams properly to free resources.
- Test with real workloads for stability.

Change Streams = Real-Time Events = Smarter Applications

MongoDB Replica Set

High Availability • Data Safety • Automatic Failover

WHAT IS REPLICA SET?

A Replica Set is a group of MongoDB servers that maintain the same data set.
It provides redundancy, high availability and automatic failover.

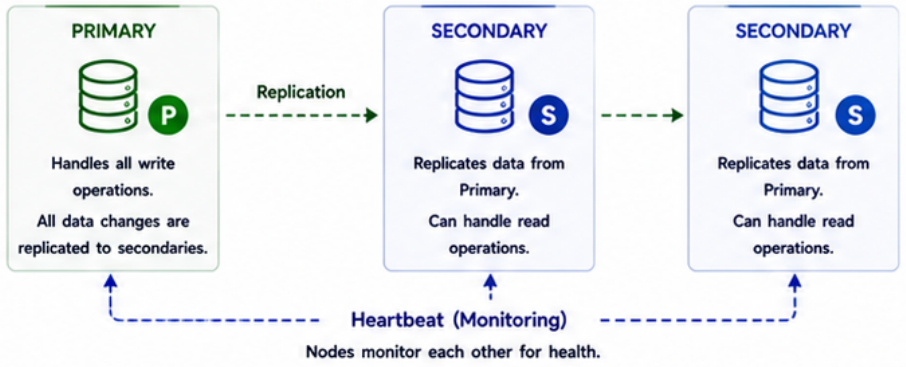
KEY FEATURES

- High Availability
- Automatic Failover
- Data Redundancy
- Scalability (Read Scaling)
- Data Safety
- Zero Data Loss (Acknowledged Writes)

PRO TIP

Always deploy an odd number of nodes (3, 5, 7) for better fault tolerance.
3 node replica set is the minimum for production.

REPLICA SET ARCHITECTURE (3 NODE EXAMPLE)



Only Primary accepts writes. Secondaries replicate data and can serve read requests.

REPLICATION FLOW

HOW FAILOVER WORKS

Election is automatic. The replica set always has one Primary.

REPLICA SET BENEFITS

High Availability
Service stays online even if a node fails.

Data Safety
Multiple copies of data prevent data loss.

Automatic Failover
Replica set elects a new Primary automatically.

Read Scaling
Read operations can be distributed to secondaries.

Fault Tolerance
System keeps running even if nodes go down.

Geo Distribution
Deploy across regions for better performance.

REPLICA SET MEMBERS

Primary
Accepts all write operations.
Sends data to secondaries.
Only one Primary at a time.

Secondary
Replicates data from Primary.
Can handle read operations.
Supports failover.

Arbiter (Optional)
Does not store data.
Helps in election process.
Usually used in 3-node setup.

DATA COPIES

Multiple copies across nodes for safety.

COMMON COMMANDS

```
// Check replica set status
rs.status()
// Initiate replica set
rs.initiate()
// View replica set configuration
rs.conf()
// Add a member to replica set
rs.add("mongodb://localhost:27018")
// Reconfigure replica set
rs.reconfig({ ... })
```

BEST PRACTICES

- Use odd number of nodes (3, 5, 7).
- Place nodes on different servers/machines.
- Monitor replica set health regularly.
- Backup your data.
- Test failover scenarios.

Replica Set = High Availability + Data Safety + Automatic Failover = Peace of Mind

MongoDB Sharding

Distributed Data • Horizontal Scaling • High Performance

WHAT IS SHARDING?

Sharding is a method of distributing data across multiple machines (shards) to handle very large datasets and high traffic.

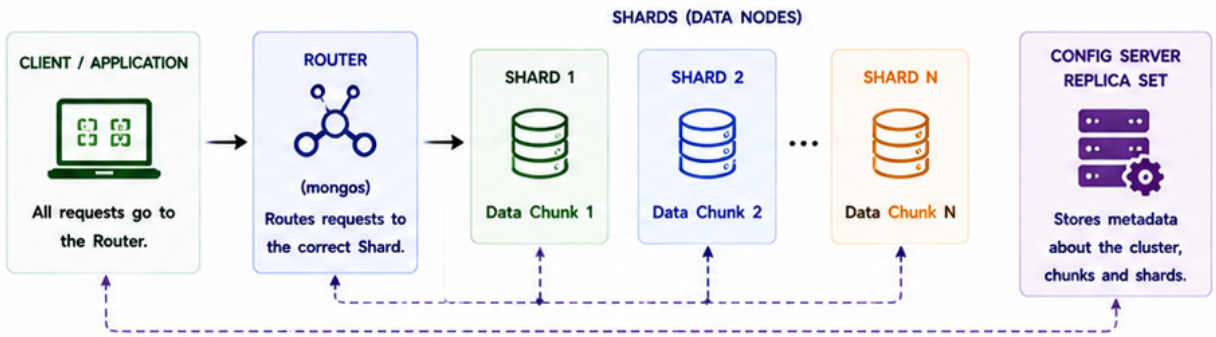
KEY BENEFITS

- Horizontal Scaling
- Better Performance
- High Availability
- Fault Tolerance
- Handle Large Datasets
- Balanced Load
- Cost Effective

PRO TIP

Sharding distributes data horizontally, not by replication but by partitioning.

SHARDING ARCHITECTURE



Router uses metadata from Config Server to know where data is located.

DATA DISTRIBUTION EXAMPLE

Collection: users

Shard Key: { userId: 1 }

Sample Data

```
{ userId: 1, name: "Amit" }
{ userId: 2, name: "Neha" }
{ userId: 3, name: "Rahul" }
{ userId: 4, name: "Pooja" }
{ userId: 5, name: "Karan" }
{ userId: 6, name: "Anjali" }
...
```

Chunk 1	Chunk 2	Chunk 3
userId: 1 - 2	userId: 3 - 4	userId: 5 - 6
Shard 1	Shard 2	Shard 3
Data: 1, 2	Data: 3, 4	Data: 5, 6

SHARDING FLOW

Sharding enables MongoDB to scale out and handle massive amounts of data with high performance.

COMPONENTS OF SHARDING

SHARD (DATA NODE)

Stores actual data. Each shard holds a subset (chunk) of the data.

ROUTER (MONGOS)

Acts as a query router. Receives client requests and forwards to shards.

CONFIG SERVER

Stores metadata about the cluster, such as chunk ranges, shard locations, and settings.

HOW SHARDING WORKS

1. Data is split into chunks based on the Shard Key.
2. Router (mongos) directs operations to the right shard.
3. Each Shard stores a subset of the data.
4. Config Servers store metadata and configuration.

SHARDING BENEFITS SUMMARY

Horizontal Scaling Add more shards to grow.	**High Performance** Faster queries and better throughput.	**Handle Large DB** Store terabytes of data easily.
High Availability Data remains available even if a shard fails.	**Load Balancing** Even distribution of data and load.	**Cost Effective** Use commodity hardware.

WHEN TO USE SHARDING?

- When data size exceeds a single server capacity.
- When you need high write/read throughput.
- When you want horizontal scaling.

BEST PRACTICES

- Choose the right shard key.
- Monitor chunk distribution.
- Keep chunks balanced.
- Avoid jumbo chunks.
- Scale gradually.

Sharding = Horizontal Scaling + Distributed Data + High Performance

MongoDB Backup & Restore

Protect Your Data • Recover Anytime • Keep Your Database Safe

WHAT IS BACKUP & RESTORE?

Backup means creating a copy of your database data so you can restore it later in case of data loss, failure or corruption.

KEY BENEFITS

- Data Protection
- Disaster Recovery
- Data Migration
- Versioning / Snapshots
- Development & Testing
- Peace of Mind

PRO TIP

Always schedule regular backups and test restore process to make sure your data is safe and recoverable.

BACKUP WITH mongodump

mongodump

Creates a binary export of your database.

Basic Syntax

```
mongodump --uri "mongodb://localhost:27017/mydb" \
  --out /backup/mydb_backup
```

Backup Examples

```
# Backup entire database
mongodump --db mydb --out /backup/mydb_backup

# Backup specific collection
mongodump --db mydb --collection users --out /backup/users_backup

# Backup with authentication
mongodump -u admin -p password --authenticationDatabase admin \\
  --db mydb --out /backup/mydb_backup
```

RESTORE WITH mongorestore

mongorestore

Restores data from a backup created by mongodump.

Basic Syntax

```
mongorestore --uri "mongodb://localhost:27017/mydb" \
  /backup/mydb_backup
```

Restore Examples

```
# Restore entire database
mongorestore --db mydb /backup/mydb_backup

# Restore specific collection
mongorestore --db mydb --collection users /backup/users_backup

# Restore with authentication
mongorestore -u admin -p password --authenticationDatabase admin \\
  --db mydb /backup/mydb_backup
```

HOW IT WORKS

1. **Create Backup**
   Use mongodump to export your data.
2. **Store Safely**
   Keep backup files in a safe location (local/remote/cloud).
3. **Restore Data**
   Use mongorestore to recover your data.
4. **Data Ready**
   Your database is back and running.

BACKUP SECURITY TIPS

- Store backups in secure locations.
- Encrypt backups if sensitive.
- Restrict access to backup files.
- Use strong authentication.
- Regularly verify backup integrity.

BACKUP OUTPUT STRUCTURE

```
/backup/mydb_backup/
└── mydb/
    ├── collection-1.bson
    ├── collection-1.metadata.json
    ├── collection-2.bson
    ├── collection-2.metadata.json
    └── ...
```

- .bson → Actual data (Binary JSON)
- .metadata.json → Index & metadata info
- System indexes and data are included

DATA RECOVERY SCENARIOS

Accidental Deletion	System Failure	Data Corruption	Migration / Upgrade
Restore the backup and recover lost data.	Recover data after server crash.	Restore clean data from backup.	Backup before migration or version upgrade.

BEST PRACTICES

- Schedule automated backups.
- Keep multiple backup copies.
- Test restore process regularly.
- Monitor backup storage space.
- Document backup & restore steps.

mongodump = Backup | mongorestore = Restore | Backup Today, Recover Anytime!

MongoDB Security

Secure Your Data • Protect Your Database • Control Access

WHAT IS MONGODB SECURITY?

MongoDB Security ensures that your data is protected from unauthorized access, data breaches and attacks by using authentication, authorization, encryption and access control.

KEY SECURITY BENEFITS

- Protects Data from Unauthorized Access
- Ensures Data Confidentiality
- Role-based Access Control
- Secure Communication
- Compliance & Audit Support
- Data Integrity & Privacy

PRO TIP

Always follow the principle of least privilege – give users only the access they need!

AUTHENTICATION

Verify the identity of users and applications

Users / Clients → **Authentication** (Validates username and password) → MongoDB Database

AUTHENTICATION METHODS

SCRAM-SHA-256	SCRAM-SHA-1	X.509	LDAP / Kerberos
Default & Recommended	Legacy Method	Client Certificate Authentication	Enterprise Authentication

AUTHORIZATION

Determine what users are allowed to do

Authenticated User → **Authorization** (Check Roles & Permissions) → Allowed Access

ACCESS IS GRANTED BASED ON ROLES & PERMISSIONS

ACTIONS CONTROLLED

Read · Write · Update · Delete · Admin

USERS

Manage database users securely

Create User → Store in System.users Collection → Authenticate & Authorize

EXAMPLE USER DOCUMENT

```
{
  "user" : "codeSardar",
  " db" : "appDB",
  "roles" : [
    { "role" : "readWrite", "db" : "appDB" }
  ]
}
```

ROLES

Predefined roles to manage permissions easily

read	readWrite	dbAdmin	userAdmin	root
Read data	Read and write data	Manage a database	Manage users and roles	Full access to all resources

You can also create custom roles for fine-grained access control.

ENCRYPTION

Protect data at rest

Encryption at Rest	Field Level Encryption
Data stored on disk is encrypted.	Encrypt specific fields in your documents.

Helps protect data even if storage is compromised.

TLS / SSL

Encrypt data in transit

Client → TLS / SSL Encryption → MongoDB Server

Ensures secure communication between clients and MongoDB servers.

ACCESS CONTROL

Control who can access what

IP Whitelisting	Network Access	Firewall Rules	Principle of Least Privilege
Allow access only from trusted IPs.	Restrict network interfaces.	Use firewall to block unauthorized access.	Give minimum access required.

SECURITY CHECKLIST

- Enable Authentication
- Use Strong Passwords
- Assign Appropriate Roles
- Enable TLS/SSL
- Encrypt Data at Rest
- Restrict Network Access
- Use IP Whitelisting
- Regularly Audit Users & Roles
- Keep MongoDB Updated
- Monitor Logs & Activity

COMMON SECURITY COMMANDS

```
# Enable Authentication
mongod --auth

# Create User
db.createUser({ user: "admin", pwd: "StrongPass123", roles: ["root"] })

# Create Role
db.createRole({ role: "readWriteRole", privileges: [
  { resource: { db: "appDB", collection: "" }, actions: ["find", "insert", "update", "remove"] }
], roles: [] })

# Grant Role to User
db.grantRolesToUser("codeSardar", ["readWriteRole"])
```

Authentication Verify Identity | **Authorization** Control Access | **Users & Roles** Manage Permissions | **Encryption & TLS** Protect Data | **Access Control** Secure Access = **Secure Database** Safe Data, Strong Future!

MongoDB Best Practices

Build Better • Run Faster • Stay Secure • Scale Smarter

1. SCHEMA DESIGN

- Design schemas based on how you query your data.
- Embed related data that is queried together.
- Use references for large or rarely accessed data.
- Avoid deep nesting.

Example

```
// Good: Embed frequently used data
{
   "name": "Aman",
   "email": "aman@example.com",
   "address": { "city": "Delhi", "zip": "110001" },
   "orders": [ { "id": 1, "total": 1200 } ]
}
```

2. INDEXES

- Index fields used in queries, sorts and filters.
- Use compound indexes wisely.
- Avoid too many indexes.
- Regularly review and clean unused indexes.

Best Practices

- Create Index on frequent queries
- Use Compound Index for multiple fields
- Monitor Index Usage

Example

```
db.users.createIndex({ email: 1 })
db.orders.createIndex({ userId: 1, createdAt: -1 })
```

3. VALIDATION

- Validate data at the schema level.
- Use $jsonSchema for strong validation.
- Prevent bad or inconsistent data.

Example (JSON Schema)

```
{
  $jsonSchema: {
    bsonType: "object",
    required: ["name", "email"],
    properties: {
      name: { bsonType: "string", minLength: 2 },
      email: { bsonType: "string", pattern: ".+@.+" },
      age: { bsonType: "int", minimum: 18 }
    }
  }
}
```

4. SECURITY

- Enable authentication (never run without it).
- Use strong passwords and roles.
- Apply least privilege principle.
- Enable TLS/SSL encryption.

Best Practices

- Authentication
- Roles & Permissions
- TLS/SSL Encryption
- Network Access Control

Tip: Never expose MongoDB to the public internet.

5. BACKUP

- Take regular backups.
- Store backups in a safe location.
- Test restore process regularly.
- Use mongodump and mongorestore.

Commands

```
# Backup
mongodump --out /backup/mongodb_backup

# Restore
mongorestore --db mydb /backup/mongodb_backup/mydb
```

6. MONITORING

- Monitor performance and resource usage.
- Use Atlas Monitoring or other tools.
- Set up alerts for critical issues.
- Keep an eye on slow queries.

Tools

- MongoDB Atlas Monitoring
- Database Profiler
- Logs & Alerts
- Third-party Tools

7. QUERY OPTIMIZATION

- Use projection to return only needed fields.
- Avoid $where and large regex queries.
- Use explain() to analyze query performance.
- Paginate large result sets.

Example

```
// Projection
db.users.find({}, { name: 1, email: 1, _id: 0 })

// Explain
db.orders.find({ status: "shipped" }).explain("executionStats")
```

8. PERFORMANCE

- Keep documents small (under 16MB).
- Use appropriate data types.
- Avoid large arrays when possible.
- Use connection pooling.

Best Practices

- Efficient Queries
- Proper Indexing
- Sharding (When Needed)
- Caching (If Possible)

9. GENERAL BEST PRACTICES

- Keep MongoDB and drivers up to date.
- Use meaningful collection and field names.
- Handle errors and exceptions properly.
- Follow consistent coding standards.

Quick Tips

- Keep it simple and consistent.
- Test in staging before production.
- Document your schema and APIs.

DO'S & DON'TS

DO'S

- Do design for your queries.
- Do create indexes wisely.
- Do validate and sanitize data.
- Do backup and test restores.
- Do monitor and optimize.

DON'TS

- Don't store unnecessary large documents.
- Don't run without authentication.
- Don't ignore slow queries.
- Don't expose MongoDB publicly.
- Don't forget regular backups.

SECURE — Protect your data and control access

OPTIMIZE — Use indexes and write efficient queries

PERFORM — Monitor, tune and scale for high performance

BACKUP — Backup regularly and test your recovery

SCALE — Design for growth and handle more data

Best Practices Today, Better Performance Tomorrow!

MongoDB Quick Reference

Essential Commands • Query Operators • Aggregation • Indexes • Best Practices

1. insertOne()

Insert a single document

```
db.collection.insertOne({
  name: "Aman",
  age: 24,
  city: "Delhi"
})
```

Result

```
{ acknowledged: true,
  insertedId: ObjectId("...") }
```

2. find()

Query documents

```
db.collection.find({
  age: { $gt: 18 },
  city: "Delhi"
})
```

Common Options

- .limit(n) – Limit results
- .sort({ field: 1 }) – Sort (1: asc, -1: desc)
- .projection({ field: 1, _id: 0 }) – Select fields

3. updateOne()

Update a single document

```
db.collection.updateOne(
  { _id: ObjectId("...") },
  { $set: { age: 25 } }
)
```

Common Operators

- $set – Set field value
- $inc – Increment value
- $unset – Remove field
- $push – Add to array

4. deleteOne()

Delete a single document

```
db.collection.deleteOne({
  name: "Aman"
})
```

Delete Options

- deleteOne() – Delete first match
- deleteMany() – Delete all matches

5. Aggregation – $match

Filter documents (like WHERE)

```
db.collection.aggregate([
  { $match: { age: { $gt: 18 } } }
])
```

Example

```
{ $match: { city: "Delhi" } }
{ $match: { age: { $gte: 21} } }
```

6. Aggregation – $group

Group documents

```
db.collection.aggregate([
  { $group: {
      _id: "$city",
      totalUsers: { $sum: 1 },
      avgAge: { $avg: "$age" }
  } }
```

Result

```
{ _id: "Delhi", totalUsers: 120, avgAge: 24.5 }
{ _id: "Mumbai", totalUsers: 80, avgAge: 26.1 }
```

7. Aggregation – $lookup

Join with another collection

```
db.orders.aggregate([
  { $lookup: {
      from: "users",
      localField: "userId",
      foreignField: "_id",
      as: "userInfo"
  } }
])
```

Result

Orders with userInfo array containing matched user documents.

8. Indexes

Improve query performance

Command	Description
db.collection.createIndex({ name: 1 })	Ascending index
db.collection.createIndex({ name: -1 })	Descending index
db.collection.createIndex({ age: 1, name: 1 })	Compound index
db.collection.getIndexes()	List all indexes
db.collection.dropIndex("name_1") }	Drop an index

9. Useful Query Operators

Operator	Meaning	Example
$eq	Equal	{ age: { $eq: 25 } }
$ne	Not Equal	{ age: { $ne: 25 } }
$gt	Greater Than	{ age: { $gt: 18 } }
$gte	Greater Than or Equal	{ age: { $gte: 18 } }
$lt	Less Than	{ age: { $lt: 30 } }
$lte	Less Than or Equal	{ age: { $lte: 30 } }
$in	In Array	{ city: { $in: ["Delhi","Mumbai"] } }
$nin	Not In Array	{ city: { $nin: ["Delhi","Mumbai"] } }

10. Projection (Select Fields)

Include / Exclude fields

```
db.collection.find(
  { age: { $gt: 18 } },
  { name: 1, age: 1, _id: 0 }
```

11. Sort

Sort documents

```
db.collection.find().sort({
  age: 1   // Ascending
})
db.collection.find().sort({
  age: -1   // Descending
})
```

12. Limit & Skip

Pagination

```
db.collection.find()
  .skip(10)   // Skip first 10
  .limit(5)   // Limit to 5
```

13. Data Types

String	"Hello"	Boolean	true / false
Int32	25	Date	ISODate("2024-05-20T10:00:00Z")
Int64	NumberLong(25)	ObjectId	ObjectId("...")
Double	25.5	Array	[1, 2, 3]
		Object	{ key: "value" }

14. Important Commands

show dbs	– Show all databases	db.collection.drop()	– Drop collection
use mydb	– Switch database	db.dropDatabase()	– Drop database
db.collection.find()	– Find all documents	db.collection.aggregate([])	– Run aggregation
db.collection.countDocuments()	– Count documents	db.currentOp()	– Show running ops
db.collection.distinct("field")	– Distinct values	db.serverStatus()	– Server status

15. Best Practices

- Use indexes for fields used in queries and sorting.
- Keep documents small (under 16MB).
- Use projection to return only required fields.
- Validate data using JSON Schema.
- Regularly backup your database.
- Monitor performance and slow queries.

16. Common Tips

- Always use appropriate data types.
- Avoid $where in production.
- Use explain() to analyze queries.
- Keep drivers and MongoDB updated.
- Use connection pooling.
- Test your queries in staging first.

Legend

- CRUD Operations
- Aggregation
- Indexes & Query
- Utilities
- Best Practices

Fast Queries • Optimized Indexes • Secure Data • Scalable Applications